AN ARCHITECTURE FOR COEXISTENCE WITH MULTIPLE USERS IN FREQUENCY HOPPING COGNITIVE RADIO NETWORKS

THESIS

Ryan K. McLean, Second Lieutenant, USAF

AFIT-ENG-13-M-34

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-13-M-34

AN ARCHITECTURE FOR COEXISTENCE WITH MULTIPLE USERS IN FREQUENCY HOPPING COGNITIVE RADIO NETWORKS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Insitute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Ryan K. McLean, B.S.C.E.

Second Lieutenant, USAF

March 2013

AFIT-ENG-13-M-34

AN ARCHITECTURE FOR COEXISTENCE WITH MULTIPLE USERS IN FREQUENCY HOPPING COGNITIVE RADIO NETWORKS

Ryan K. McLean, B.S.C.E.
Second Lieutenant, USAF

Approved:

Mark D. Silvius Mark D. Silvius, Major, USAF, PhD (Chairman)	5 Mar 2013 Date
Kenneth M Hopkinson Kenneth M. Hopkinson, PhD (Member)	7 Mar 2013 Date
Mary Y Lanzerotti Mary Y. Lanzerotti, PhD (Member)	8 March 2013 Date

AFIT-ENG-13-M-34

Abstract

The radio frequency (RF) spectrum is a limited resource. Spectrum allotment disputes stem from this scarcity as many radio devices are confined to a fixed frequency or frequency sequence. One alternative is to incorporate cognition within a reconfigurable radio platform, therefore enabling the radio to adapt to dynamic RF spectrum environments. In this way, the radio is able to actively observe the RF spectrum, orient itself to the current RF environment, decide on a mode of operation, and act accordingly, thereby sharing the spectrum and operating in more flexible manner. This research presents a novel framework for incorporating several techniques for the purpose of adapting radio operation to the current RF spectrum environment. Specifically, this research makes six contributions to the field of cognitive radio: (1) the framework for a new hybrid hardware/software middleware architecture, (2) a framework for testing and evaluating clustering algorithms in the context of cognitive radio networks, (3) a new RF spectrum map representation technique, (4) a new RF spectrum map merging technique, (5) a new method for generating a random key-based adaptive frequency-hopping waveform, and (6) initial integration testing toward implementing the proposed system on a field-programmable gate array (FPGA).

To my beautiful wife: Simply put, I really couldn't have done this without you.

Acknowledgments

I would first like to thank my advisor and thesis committee chair for his unrelenting support, guidance, and critical review. My concept of scientific research and the research process was challenged and expanded while at AFIT, and I am duly grateful for his encouragement in that endeavor. He has also served as a valuable mentor in my first active duty tour. Additionally, I owe sincere thanks to my committee: to Dr. Hopkinson for his guidance in algorithm and conference selection, and to Dr. Lanzerotti for her keen insight into thorough paper formulation and editing. Any errors in this document are my own.

Finally, I extend my and my committee's thanks to AFRL/RWYE for their continuing support of this research, including lab access, test data generation, experiment guidance, and project funding.

Ryan K. McLean

Table of Contents

Page

Page

Page

List of Figures

Figure Page

List of Tables

List of Acronyms

Acronym	Definition
AFH	Adaptive Frequency Hopping
AFIT	Air Force Institute of Technology
AFRL	Air Force Research Laboratory
AHS	Adaptive Hopset Selector
BEE2	Berkeley Emulation Engine 2
BRAM	Block RAM
CE	Cognitive Engine
CR	Cognitive Radio
CRN	Cognitive Radio Network
DAFH	Dynamic Adaptive Frequency Hopping
DARPA	Defense Advanced Research Projects Agency
DSA	Dynamic Spectrum Access
DYSE	Dynamic Spectrum Emulator
EDK	Embedded Development Kit
FASU	Frequency Agile Spectrum Usage
FCC	Federal Communications Commission
FFT	Fast Fourier Transform
FH	Frequency Hopping
FPGA	Field-Programmable Gate Array
GPS	Global Positioning System
HELLO	*(Not an acronym)*
HF	High Frequency
ICSS	Intra-Cluster Spectrum Similarity

Acronym	Definition
IEEE	Institute of Electrical and Electronics Engineers
IP (core)	Intellectual Property
IP (address)	Internet Protocol
ISM	Industrial-Science-Medicine
KUAR	Kansas University Agile Radio
KTA	Key Technology Area
MAC	Media Access Control
OLSR	Optimized Link State Routing
OODA	Observe-Orient-Decide-Act
PCA	Potential Capability Area
PCSR	Public Safety Cognitive Radio (Virginia Tech)
PU	Primary User
RAM	Random Access Memory
REM	Radio Environment Map
RF	Radio Frequency
SA	Situational Awareness
SCF	Service Core Function
SDR	Software-Defined Radio
SINCGARS	Single Channel Ground and Airborne Radio System
SINR	Signal to Interference and Noise Ratio
SISO	Single Input Single Output
SU	Secondary User
TOMC	Totally-Ordered Multicast
UHF	Ultra High Frequency
UNII	Unlicensed National Information Infrastructure

Acronym	Definition
UWB	Ultra-wideband
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
WARP	Wireless Open Access Research Platform
WLAN	Wireless Local Area Network
WPAN	Wireless Personal Area Network
WRAN	Wireless Regional Area Network
WPAFB	Wright Patterson Air Force Base

AN ARCHITECTURE FOR COEXISTENCE WITH MULTIPLE USERS IN FREQUENCY HOPPING COGNITIVE RADIO NETWORKS

1 Introduction

TODAY's radio frequency (RF) spectrum is increasingly congested. In America, the Federal Communications Commission (FCC) partitions the RF spectrum into frequency ranges based on the needs and capabilities of radio equipment. Originally, the FCC performed this partitioning according to input from public hearings. Over time, spectrum allocation became the product of systems such as lotteries and spectrum auctions. Currently, the pervasiveness of equipment that uses the RF spectrum (radio and television stations, cell phones, wireless internet, etc.) demands more flexibility of the spectrum allocation process. In the future, the ever-expanding wireless footprint implies that devices will need to adapt their transmission and reception capabilities to the RF spectrum in which they communicate. This practice is known as "dynamic spectrum access" (DSA). However, even with an adaptive communication technique, the RF spectrum's density (especially the overlapping of spread-spectrum transmissions) will still likely detract from an adaptive waveform's transmission.

Additionally, some spectrum assignments are applied to all times during the day, yet many are only used consistently for brief periods of time. Frequencies which *may* be used at any given time by a specific user, but which *are* used only periodically, significantly contribute to wasted spectrum assignment. This issue is extensible to the military's need for radios which can function regardless of their geographic location, the time of day, and the local spectrum policy. Were military (or civilian) radios able to operate in unused spectrum

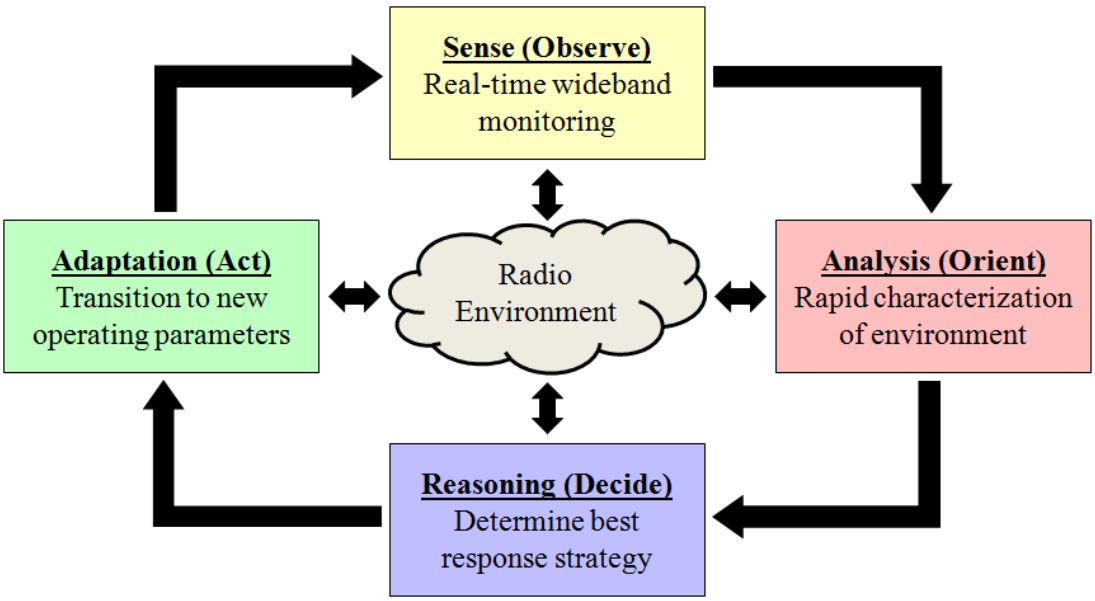


Figure 1.1: The OODA loop as a "cognitive cycle" [3].

during times of limited actual usage, the spectrum could be used much more efficiently. This research presents an architecture through which this need can be met.

It is believed a cognitive radio (CR) can be used to implement DSA in solving the aforementioned questions as related to RF spectrum congestion and spatial/temporal policy conflicts. CRs are a subset of the now-common radio implementation, the software-defined radio. The CR's power is twofold. First, by using a field-programmable gate array (FPGA), the CR becomes a high-speed, circuit-based yet reconfigurable device [1]. (For this reason, some refer to CRs as "firmware-defined radios"). Second, CRs implement Colonel John Boyd's famed Observe-Orient-Decide-Act ("OODA") loop as a means by which to make informed, adaptive decisions on transmission and reception policy. In summary, the CR is DSA's enabling technology [2].

Colonel Boyd's OODA loop models the initial problem set. First, a radio senses its local RF spectrum (observation). Next, the software contained on the radio constructs a map of the RF spectrum (a radio environment map, or REM) from which to use available

frequencies (orientation). Third, the radio assembles a randomized frequency hopping pattern (the *hopset*), for distribution to neighboring radios (decision). Finally, the radio uses the hopset for adaptive communication (action). There are various namings, interpretations, and implementations of this model, such as that of [3] (e.g., the "cognitive cycle"), but each distills down to the OODA loop. Figure 1.1 shows how the OODA loop maps directly to the interpretation from [3]. The mapping of colors to OODA loop functions remains constant in the related diagrams throughout this document.

Figure 1.2 shows how the proposed system is follows the OODA model over time in the presence of interference. It is worthwhile to note that this system is not focused on the transmission or reception of specific data—only adapting to the current RF environment. At period A, the system constantly observes the environment. Period B shows notional expected performance while the spectrum is changing, the existing adaptations are under duress, and the radio is reorienting itself. Period C is when the system must decide how to react to the new RF environment. The system (re)acts using its decision at period D and, finally, resumes observation at some improved level of performance (three feasible outcomes are shown).

The natural extension of this process is implementing it in a cognitive radio network (i.e., two or more). In this case, the observation step involving spectrum sensing expands to include spectrum observation dispersion within the network. If those radios which must communicate are aware of the RF spectra at those other nodes within the network, then observation is complete within the network.

1.1 Motivation

1.1.1 Air Force Technology Horizons 2010.

The vision in [4] delivers a two-decade (2010-2030) outlook of the Air Force's current technological capabilities, an assessment of future challenges, and a plan for the molding

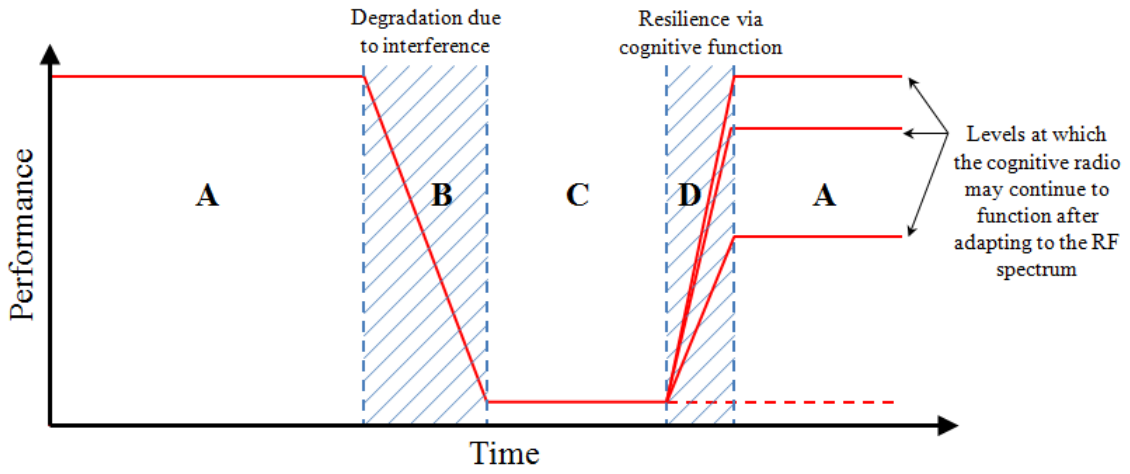


Figure 1.2: Expected system operation.

of Air Force technology to meet such challenges. Produced by the office of the Air Force Chief Scientist, it provides a high level review to the service's senior staff, specifically the Secretary of the Air Force and Chief of Staff of the Air Force. "Technology Horizons" presents 12 Air Force "Service Core Functions" (SCFs), analogous to "strategic"-level functions needed by the service to accomplish its missions. Also identified are 30 "Potential Capability Areas" (PCAs). PCAs are defined as "credibly achievable within the time horizon addressed by this study." Additionally, each PCA is mapped to each SCF such that every SCF is supported by multiple PCAs. Under the PCAs, and at a finer level of granularity, the document also lists 110 "Key Technology Areas" (KTAs). KTAs compose a large, overlapping set of those technologies that make each PCA possible. In paraphrase, these are the focus areas the Air Force believes it can and will technologically conquer by the year 2030 [4].

PCA number seven is "Frequency Agile Spectrum Utilization" (FASU). Within FASU is Dynamic Spectrum Utilization (DSA), the core tactical goal of this thesis. The fact that the Air Force has identified FASU, and by extension, DSA, as a major tenet of the service's near-, mid-, and long-term technical superiority calls for additional research, development,

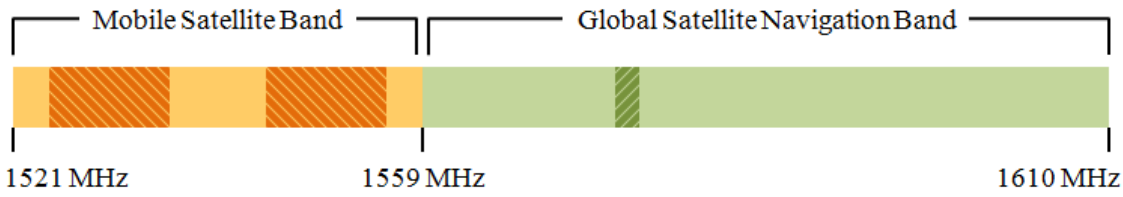


Figure 1.3: Conflicting LightSquared and GPS spectrum assignments.

and fielding of systems in this key area. In the context of [4], FASU is featured in five PCAs. Of larger impact is the high number of SCFs to which the FASU PCA maps: PCA number seven maps to 11 out of 12 SCFs; of all 30 PCAs, only five map to an equal or greater amount of SCFs. The bottom line is this: FASU and DSA are projected to have a *significant* impact on Air Force strategy and operations over the next two decades.

1.1.2 LightSquared versus the FCC.

Spectrum access and sharing issues are also of relevance in the civilian sector. In recent years, the company LightSquared has sought to build a nation-wide wireless broadband network using previously vacant spectrum (Figure 1.3, darkened within the Mobile Satellite Band) near the GPS band (Figure 1.3, darkened within the Global Satellite Navigation Band). In fact, the effort was originally sanctioned (and even propelled) by the FCC. It was eventually realized through testing, though, that LightSquared's spectrum allotment and the energy generated within that spectrum would bleed over into the global satellite navigation band. Specifically, the GPS sub-band would be impacted, if not drowned out, in certain locations [5].

As a result of this conflict, the FCC directed LightSquared to form a working group comprised of LightSquared, government, and industry GPS engineers. The working group's decision was issued in a report in June 2011 with the declaration that the current course of action would indeed degrade GPS receiver performance. More damning was a comment from the group's aviation committee: "For the originally defined LightSquared

spectrum deployment scenarios, GPS-based operations are expected to be unavailable over entire regions of the country at *any normal operational aircraft altitude*". In other words, in many areas aircraft flying below 2,000 feet above ground level would have little or no GPS guidance. This is particularly dangerous because many aircraft, from general aviation to airliners to military aircraft, use a high-precision version of GPS to take off and land. Degraded GPS in these environments can be deadly, especially if it is unexpected.

The obvious source of contention is spectrum allocation policy as it pertains to governing transmitters. However, [5] poses the following question: Is it time for the FCC to focus on *protecting receivers*? Unfortunately, the LightSquared/FCC debate is too advanced redesign the underlying hardware, but in future and with receiver-oriented regulations, a situation such as the LightSquared incident could be resolved on the drawing board. For example, if GPS receivers were protected, the issues would be oriented toward the end-user from the very start of system design—it is reasonable to assume the LightSquared issue could have been avoided early in the process by designing around receiver protections. Although there are no FCC regulations in place to protect receivers, this research is a start toward solving the problem of dynamic receiver behavior in the presence of interference.

The LightSquared incident is a clear example of the need for DSA and FASU on a national infrastructure scale. Furthermore, CR is an ideal method for implementing both concepts. Given a CR's natural ability to avoid spectrum conflicts, it becomes a highly attractive solution to conflicts ranging from low-profile, low-impact personal networks to high-profile, high-impact situations such as the predicament of LightSquared.

1.1.3 Tactical DSA.

The Defense Advanced Research Projects Agency (DARPA) has also identified the need for DSA in tactical scenarios. Currently, ground-based troops have no means for detecting spectrum usage for the purpose of avoiding such interference. In theater, troops

are often faced with spectrum congestion. In a tactical environment where a mission is at stake and lives are at risk, arguably the worst problem for a commander is a breakdown or total loss of communication with his troops. DARPA seeks to alleviate this issue by providing commanders and their troops with a means for identifying occupied and unoccupied RF spectrum. This capability does not focus on the content of transmissions, but solely on classifying the energy present in the local RF spectrum [6, 7].

Given a digital analysis of the spectrum, this research presents a method in which a map of the radio spectrum intelligible by frequency-hopping hardware is produced. The representation is not tailored for visual inspection or human interface as would be DARPA's proposed map. It does, however, give the components implemented in this research the capability to identify unoccupied portions of the RF spectrum for the purpose of avoiding unnecessary congestion and interference.

Additionally, DARPA previously researched a new method for taking advantage of unused RF spectrum–conceivably for use with such a map primarily for public DSA [8]. The neXt Generation (XG) communication program was proven in 2006 in a six-node network to be able to opportunistically use otherwise wasted spectrum [9]. The proposed system has the same basic function regarding the ability to use open spectrum space. In addition to building on the XG radio's functionality and network size, this research involves a fully-scalable network (to be tested in this work at 100 nodes) and a frequency-hopping waveform in place of a single-frequency operation.

1.1.4 Cognitive Radio Network Reliability.

If a radio is truly cognitive, there is a well-defined set of parameters by which the radio evaluates its situation and reacts in a way that enables it to continue operation using a new configuration. We believe reliability in a cognitive radio network is the result of effective resource usage after it adapts to its environment. Our cognitive radio uses a frequency hopping waveform for communication. Therefore, the system's cognitive functionality

stems from its ability to evaluate occupied and unoccupied frequencies within a network, form an adaptive hopset, and use the selected waveform. By implementing an adaptive process, the CR becomes a direct solution to the questions of FASU and DSA in that is reconfigurable to multiple frequencies and can access the spectrum in a self-propelled manner.

Reliability in our system, then, is a function of the number of frequencies from which the radio selects its hopset. A large set of open frequencies allows the radio to traverse more of the RF spectrum, thereby decreasing the likelihood of significant data loss due to interference. It is expected geographically partitioning a CRN yields groups, or *clusters*, with similar observed RF spectra. This experiment investigates which clustering heuristic, if any, partitions the network such that overall network complexity is reduced for the purpose of minimizing extraneous network traffic when transmitting data (i.e., spectrum maps, waveforms, actual data, etc.). If network complexity is reduced, it is expected the network becomes more reliable as data is transmitted more efficiently and with fewer collisions and/or delays.

1.2 Problem Statement

The following operations govern the functionality of the whole system. Brief operation descriptions are given here, while full explanations are presented in Section 3.1.2. Each operation corresponds to a state of the same label in Figure 1.4, where "network partitioning" and "waveform" map to implementing "clustering" and "hopset," respectively, in this research. Additionally, each operation maps to the color-corresponding OODA function in Figure 1.1. This coloring scheme is continued through the remainder of the document.

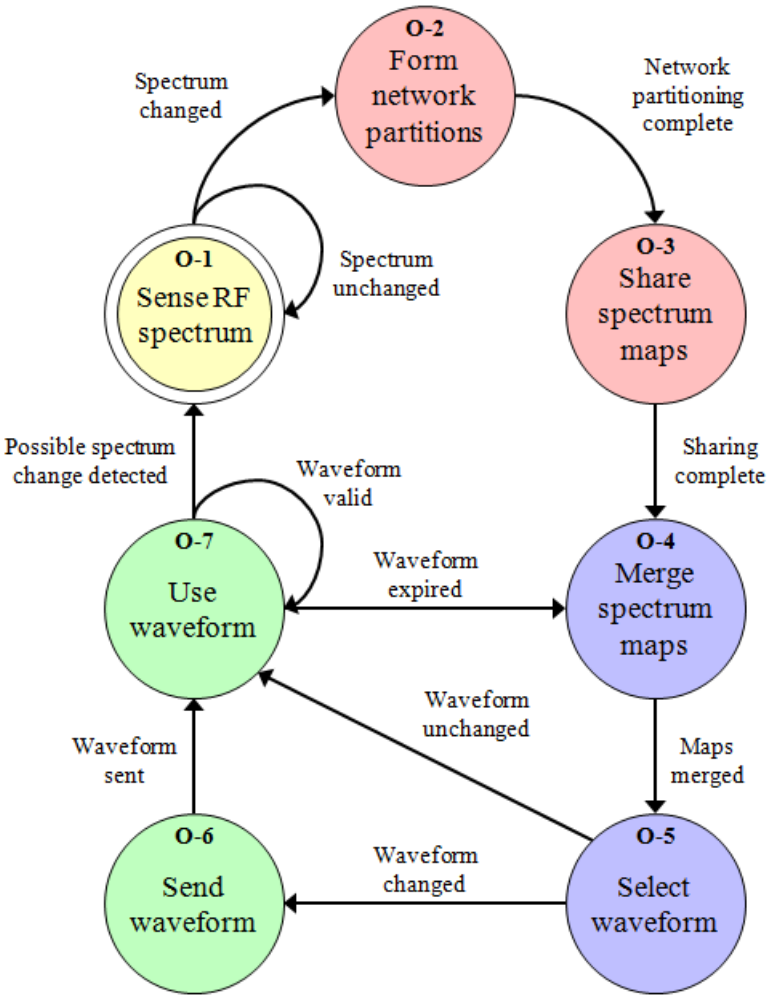


Figure 1.4: System function.

- **O-1. Sense RF spectrum.** Detect the presence of other spectrum users.
- **O-2. Form network partitions.** Partition the network into sub-networks based on geographic location.
- **O-3. Share spectrum maps.** Distribute spectrum knowledge throughout the network.
- **O-4. Merge spectrum maps.** Fuse the sensed data for optimal waveform selection.

- **O-5. Select waveform.** Generate a waveform common to all nodes within a sub-network.
- **O-6. Send waveform.** Distribute waveform information to all nodes within network.
- **O-7. Use waveform.** Engage in normal communication using the adaptive waveform.

The issues identified in Section 1.1 represent a sampling of the issues currently recognized within the DSA community. It is expected a framework that accomplishes the preceding seven steps is a step toward answering some of these questions. This research proposes such a framework in the form of a middleware architecture and implements several key components. Succinctly, the goal is to answer the following question: *How can we implement a frequency hopping cognitive radio network for coexistence with multiple users?*

Within the above problem statements, several terms are defined:

- **Frequency hopping.** Devices must be able to use a frequency hopping waveform that utilizes multiple frequencies for discrete periods of time.
- **Cognitive radio network.** Two or more radios which can operate according to the OODA loop, specifically the architecture proposed in this work, form a cognitive radio network.
- **Coexist.** Cognitive radios are designed for the purpose of avoiding other energy sources in the RF spectrum.
- **Spectrum users.** It is highly likely that other devices similar to the one in this research also are capable of using the same RF spectrum usable by the proposed system.

1.3 Research Contributions

This research makes six contributions to the field of cognitive radio:

1. The framework for a new hybrid hardware/software middleware architecture (see Figure 3.1);
2. A framework for testing and evaluating clustering algorithms in the context of cognitive radio networks (see Figure 3.1, **O-2** and Figure 3.5);
3. A new RF spectrum map representation technique (see Figure 3.6);
4. A new RF spectrum map merging technique (see Figure 3.1, **O-4** and see Figure 3.22);
5. A new method for generating a random, key-based adaptive hopset frequency hopping waveform (see Figure 3.1, **O-5** and see Figure 3.23); and
6. Initial integration testing toward implementing the proposed system on a field-programmable gate array (FPGA) (see Figure 4.14).

It is expected that these contributions form the foundation for fulfilling the original architecture, where the eventual result is a deployable prototype *ad hoc* network of cognitive radios which operate in the presence of additional spectrum users.

1.4 Thesis Organization

Related existing work and background information on implemented components is summarized in the next chapter. Components implemented in this project are validated in the Methodology chapter and analyzed in the Results chapter. Conclusions are presented in the fifth chapter, and future work is detailed in the last chapter. Appendices containing additional figures and software code are at the end of this document.

2 Related Work

This work combines the results of previous related but independent endeavors, specifically spectrum mapping techniques, network clustering, FPGA-based cognitive radio implementations, and adaptive frequency hopping (AFH). This chapter addresses original work in these areas and this work's contributions to each area.

2.1 Cognitive Radio

The ever-crowded RF spectrum demands a utility by which spectrum may be used more efficiently by more users for less cost. Primary users purchase spectrum bands for unrestricted usage. Such users do not always use their allotted spectrum. Secondary users (SUs), on the other hand, do not possess ownership of any RF spectrum bands and are relegated to the unlicensed bands (Industry-Science-Medicine (ISM), Unlicensed National Information Infrastructure (UNII), etc.). Given the permeation of radio-centric devices (cellular telephones, wireless internet, Bluetooth, Zigbee, etc.) into typical daily functions, secondary users are highly prone to inadvertent interference from other secondary users. This realization begs the question: How can secondary users with their limited spectrum assignments efficiently use underutilized spectrum allocated to primary users?

Past systems designed for resilience in an interference- or jamming-prone environment paved the way for today's advanced radio systems. For example, radios such as the military's SINCGARS and HAVE QUICK hop frequencies based on a user-supplied key, but the hopping sequence is independent of the surrounding spectrum. Legacy adaptive frequency hopping radios date back to the early 1990s, but only in the past decade has the requisite commercial technology made such systems feasible in non-military applications. The need for pervasive adaptable radios has long been recognized, and the supporting research and competencies are now coming to fruition.

Enter the cognitive radio. A progression of software-defined radio (SDR), a cognitive radio boasts the technologies necessary to use large portions of RF spectrum designated for primary users as the transmission medium for secondary users. One of the largest attractions to cognitive radios is that they are designed to adapt to the needs of spectrum regulators, network operators, and user objectives [10]. This promise has driven explosive growth in just the past five years. For example, in [10], it is mentioned that an internet search on the term "cognitive radio" in 2009 yields 138,000 hits–triple the number in 2006. A similar search today, however, gives 6,820,000 hits–*nearly 50 times* the number four years ago. Clearly, cognitive radio has taken hold.

Cognitive radio is defined by the SDR Forum [11] and the Institute of Electrical and Electronics Engineers (IEEE) 1900.1-2008 [12] working group as follows:

1. "Radio in which communications systems are aware of their environment and internal state, and can make decisions about their radio operating behavior based on that information and predefined objectives. The environmental information may or may not include location information related to communication systems.

2. "Cognitive radio (as defined in 1) that uses SDR, adaptive radio, and other technologies to automatically adjust its behavior or operations to achieve desired objectives."

When fueled by popularity among engineers and businesses, formal adoption within the scientific community, and maturation of key technologies, the cognitive radio is quickly becoming the means to achieve what is needed: a smarter radio. This work follows the above definitions for building and implementing a cognitive radio

2.1.1 An Increasingly Smarter Radio.

A popular and growing field, cognitive radio has seen steady advancement, most notably in the past five years. Efficient spectrum usage, to include dynamic allocation

techniques and sharing schemes, drives most of this research for the purpose of allowing secondary users to avoid primary users [3]. In their journal article, the authors of [3] cover the technological advances in cognitive radio. These advances are addressed in Tables 2.1, 2.2, 2.3, and 2.4 along with the projected level of contribution of this research to each area (Primary, Secondary, or Not Applicable [N/A]). Primary areas receive a direct contribution from this work, secondary areas are those on which this work is formed but do not receive any new contribution from this work, and irrelevant areas are not the focus of any contribution in this work. An argument is presented for each for each irrelevant item as relevant items are further discussed throughout the document.

Each fundamental advance is relevant to this research, due in large part to the broadness of each area. However, spectrum sensing is assumed to be an existing function prior to implementing the proposed architecture. Cognitive capability (i.e., adaptive hopset selection) and dynamic network reconfiguration comprise the core contributions of this research.

Interference temperature is a non-issue in this research as it is assumed each radio can innately and accurately identify spectrum whitespace and extract signal contents. Likewise,

Table 2.1: Recent advances in CR fundamentals.

Area	**Advances**	**Contribution**
CR characteristics	New communications and networking paradigms, cognitive capability, etc.	Primary
CR functions	Spectrum sensing/analysis, management/handoff, and allocation/sharing	Secondary
Network architecture and applications	Spectrum broker entities, dynamic network reconfiguration, etc.	Primary

Table 2.2: Recent advances in spectrum sensing and analysis.

Area	Advances	Contribution
Interference temperature	Tolerable interference levels	N/A
Spectrum sensing	Energy detection, feature detection, match filtering/coherent detection, etc.	Secondary
Cooperative sensing	User selection, decision fusion, and efficient information sharing, and distributed cooperative sensing	Primary

hardware-based sensing and energy detection are not direct contributions, but determining how to use said whitespace is a large part of forming the adaptive hopset. Further, doing so within the CRN and in a distributed, collaborative fashion is key to the architecture's implementation.

Adaptive clustering and hopset selection are the central thrusts of this research, so licensed spectrum sharing, power control, game theory in spectrum sharing, and cooperation enforcement receive no contribution. Spectrum handoff, cognitive relay, and routing are all targets of continued development but are not immediately addressed by this research. Similarly, this work makes use of advances in the CR MAC layer and lays the groundwork for a control channel and related management schemes, but does not contribute to either area. Security in this network is a noble consideration and one that must be investigated. This research does not, however, address security in order to maintain public distribution standards.

While this research does use existing standards and regulations as guides, contributions to standards are not central to the work. Research implementations, however, are highly relevant. In addition to being a research implementation in itself, this work builds on several existing implementations.

Table 2.3: Recent advances in dynamic spectrum allocation and sharing.

Area	Advances	Contribution
Licensed spectrum sharing	Spectrum underlay and overlay	N/A
Media Access Control (MAC) in CRNs	MultiMAC protocols and applicability of the MAC layer to spectrum access schemes	Secondary
Spectrum handoff	Suspending transmission when a PU reappears and resuming operation via contingency planning	Secondary
Cognitive relaying	Spectrum-opportunistic packet forwarding within a CRN	Secondary
Spectrum sensing and access	Partially observable Markov decision process (POMDP), optimality of myopic policies, and inclusion of SU residual energy and buffer state in POMDP	Primary
Power control in a CRN	Impact of transmission power on available spectrum, dynamic programming for optimal power and rate control, collaborative power sensing, etc.	N/A
Control channel management	Cluster-specific control channel based on common channels, "swarm-based" selection according to whitespace analysis, etc.	Secondary
Distributed spectrum sharing	Independent action for fair resource sharing, time-spectrum blocking	Primary

	(similar to GSM cellular technology), and multi-agent learning for resource management	
Spectrum sharing game	Game theory in spectrum resource management, no-regret learning while considering cooperative/non-cooperative users, auction mechanisms, etc.	N/A
Routing in a CR network	Accounting for lack of channel commonality, multiple channel switches along a path, variable channel switching delay, etc.	Secondary
Cooperation stimulation and enforcement	Assuming not all nodes are unconditionally cooperative in system design, cooperation degree metrics, credit mechanisms, etc.	N/A
Security in CRNs	Localization-based defense, spectrum sensing data falsification attacks, induced SU/PU interference, etc.	N/A

Table 2.4: Recent advances in spectrum sensing and analysis.

Area	Advances	Contribution
IEEE 802.22	Relatively new standard governing cognitive radio and secondary user access in the TV spectrum [13]	Secondary
IEEE P1900.1	Standards series centered around spectrum management and next-generation radios [12]	Secondary
Research implementations	Berkeley Emulation Engine 2 (BEE2) [14], Public Safety Cognitive Radio (PSCR) at Virginia Tech [15], and Rugers University's Open Access Research Testbed for Next-Generation Wireless Networks (ORBIT), etc.	Primary

This research takes advantage of and builds on numerous recent cognitive radio advances including fundamental characteristics, dynamic network architectures, cooperative sensing, spectrum access, distributed spectrum sharing, and existing research implementations. These advances and many others are addressed in [3], providing a broad survey from which to launch into the actual research process. Contributions to relevant areas are further validated in this chapter and the Methdology.

2.1.2 Governing Standards.

As a technology growing in both conceptual popularity and practical implementation, new standards are needed to support CR use [16]. National governing organizations including the American Federal Communications Commission (FCC) and the British Office of Communications (Ofcom) are already considering adopting laws to govern

cognitive radio usage–specifically, the ability of SUs to use fallow RF spectrum allocated to PUs. Additionally, multiple organizations including the International Telecommunications Union - Radio Sector (ITU-R), SDR Forum, and the IEEE have begun work on CR-centric standards. Namely, the IEEE has already published two standards as they relate to CR, SCC41 (formerly P1900) and IEEE 802.22.

The authors of [16] also note that the emergence of such standards is particularly useful when multiple users are competing for the same RF spectrum whitespace. When two users compete in this fashion, they often are forced by such standards to share channels with applicable users. This is known as *self*-coexistence and is implicitly applicable to this research as the proposed system blindly adapts to unavailable spectrum. In this way, this work's CRN performs self-coexistence naturally as new users make their presence in the spectrum.

A detailed examination of the IEEE 802.22 standard is presented in [13]. The sticking point of this article is that IEEE 802.22 is the first *worldwide* standard based on cognitive radio. Although the article is from 2005, it serves to highlight the growing prominence of CR as the solution to spectrum overcrowding. According to the article, in 2005 it was estimated that at any given time, only 5.2% of the RF spectrum is usable by SUs–even though PUs often leave their frequencies dormant and unused. The authors also define the standard's relation to existing network standards (see Figure 2.1), where the outermost ring is the proposed CR domain. Although not designed to operate at higher throughput rates (i.e., 18-24 Mbps compared to IEEE 802.11n at over 100 Mbps), IEEE 802.22 provides the key functionality of allowing SUs to opportunistically operate in PU-designated (but often vacant) spectrum.

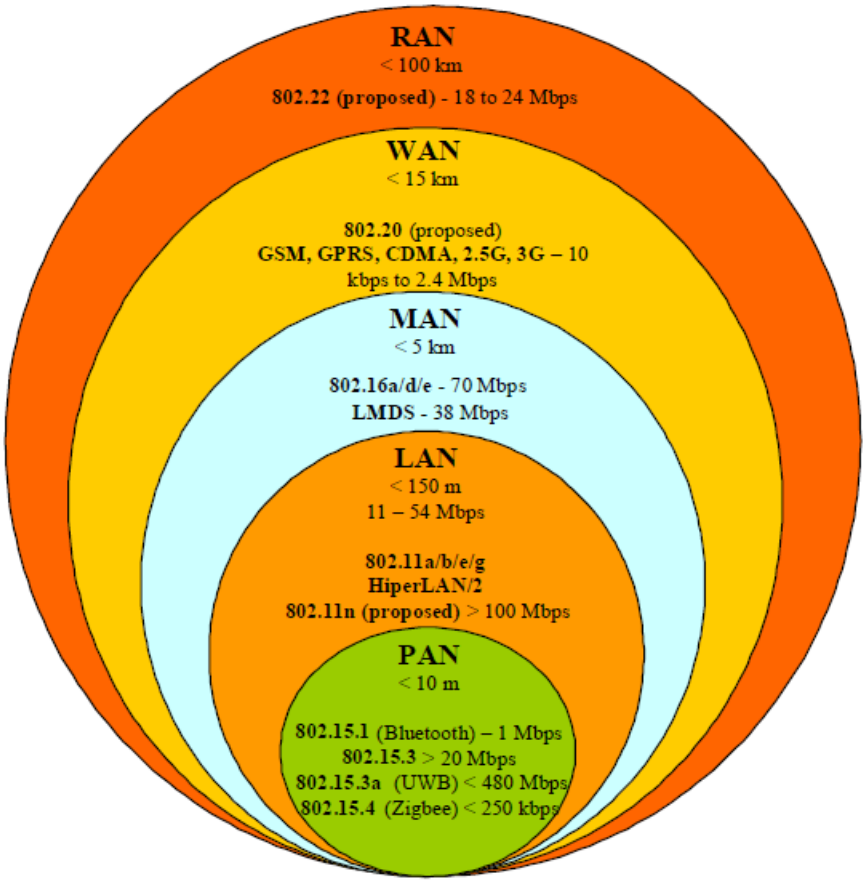


Figure 2.1: Relation of IEEE 802.22 to other IEEE network standards [13].

2.2 Adaptive Frequency Hopping (AFH)

2.2.1 Legacy Frequency Hopping Systems.

2.2.1.1 Single Channel Ground and Airborne Radio System (SINCGARS).

SINCGARS is a cryptographically-keyed frequency hopping radio originally intended for ground forces. The radio can operate over 2,320 channels from 33 to 80 megahertz (MHz) and has modular construction to ease the upgrade process. The operator is able to adjust the radio's functionality using a collection of controls on the front of the device, shown in Figure 2.2 [17, 18].

Figure 2.2: Standalone SINCGARS radio [19].

2.2.1.2 HAVE QUICK.

HAVE QUICK is the U.S. military's standard ultra-high frequency (UHF) radio standard. Implemented on a host of airborne platforms, HAVE QUICK is a sophisticated frequency hopping technology. Similar to SINCGARS, it uses a cryptographic key to produce a frequency hopping sequence that is chronologically synchronized among all users for a given time period. Figure 2.3 shows the cockpit panel mount in a NATO F-16 aircraft [18, 20]

Figure 2.3: HAVE QUICK panel mount in an F-16 [20].

2.2.1.3 High Frequency (HF) AFH.

Adaptive frequency research has been ongoing since the early 1990s. In their 1993 MILCOMM paper, the authors of [21] introduce a scheme for AFH based on observing channel states and generating a frequency hopping sequence that uses "good" channels more often than "bad" channels. Also mentioned is the stipulation that available frequencies are not selected uniformly. This system is modeled in Figure 2.4. Later, in 1996, another approach for AFH involving correlated frequency hopping. While not overtly focused on AFH, [22] does highlight the need for improved frequency hopping measures for the purpose of avoiding "limited bandwidth or intentional jamming"–both of which are addressable using CR technology.

2.2.1.4 Summary.

SINCGARS and HAVE QUICK represent modern U.S. and NATO frequency hopping radio standards. However, each implementation is not adaptable to the current RF spectrum

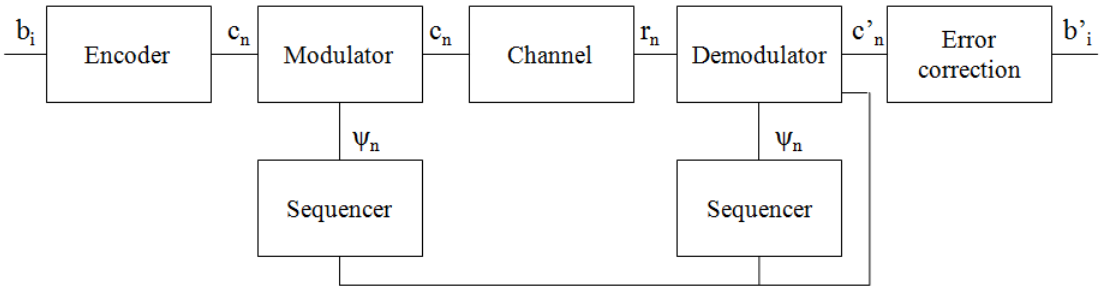


Figure 2.4: Early AFH system [21].

and, even if the RF environment is known, requires a "human-in-the-loop" to manipulate the radio settings to fit such an environment. By definition, cognitive radio has the innate ability to sense the spectrum, orient its operation, decide on a course of action, and perform the appropriate actions. A human operator must engage the same process, but because a CR can autonomously tweak the required knobs and meters to fit the existing RF environment, it can operate much faster and at the expense of little or no valuable manpower. AFH systems began to emerge in the early 1990s, but not in the context of a truly *cognitive* radio. As compared to legacy systems such as SINCGARS and HAVE QUICK, CR allows for a quicker, more precise method for implementing a frequency hopping radio. The AFH iteration presented in this research is implemented as a CR with all necessary automation.

2.2.2 Bluetooth (802.15.1) & WLAN (802.11).

Adaptive frequency hopping is an emerging trend in CR networks. As they are already able to sense spectrum availability, CR networks are natural candidates for expanding the adaptive envelope to include entire hopsets in addition to single frequencies. An example of AFH in practice is the current Bluetooth implementation. Whereas the early standard was spread out across 79 of the possible 83.5 ISM channels, Bluetooth AFH versions need only hop over 15 channels. This allows for much less interference to other devices operating in the same band, including WLAN (802.11) and ZigBee (802.15.4) [23].

Adaptive frequency hopping is an emerging trend in CR networks. As they are already able to sense spectrum availability, CR networks are natural candidates for expanding the adaptive envelope to include entire hopsets in addition to single frequencies. An example of AFH in practice is the current Bluetooth implementation. Whereas the early standard was spread out across 79 of the possible 83.5 ISM channels, Bluetooth AFH versions need only hop over 15 channels. This allows for much less interference to other devices operating in the same band, including WLAN (802.11) and ZigBee (802.15.4) [23, 24].

2.2.3 Dynamic Adaptive Frequency Hopping.

The authors of [25] propose the dynamic adaptive frequency hopping (DAFH) algorithm in wireless personal area networks (WPANs). They define DAFH as a general method that enables the FH-networks in the unlicensed band to avoid mutual interference in a distributed manner. Their approach is similar to ours in that they determine available frequencies and select a hopset using a common seed for pseudorandom hopset generation. The authors also implement per-subnet band division, i.e. the overall band is partitioned such that every neighboring subnet uses a separate frequency sample space to avoid collisions between hopsets. We currently intend to simply ensure no two subnets will share the same hop at the same time, but the technique in [25] is certainly worthwhile to explore for future development.

2.2.4 Summary.

Adaptive frequency hopping (AFH) forms the core basis by which the proposed system reliably and unpredictably avoids interference. Frequency hopping applications currently drive military radio communication systems, but none hops according to adaptation to the RF environment. Commercial applications such as Bluetooth have begun implementing simple packet loss-induced AFH schemes, but such methods continue to use occupied frequencies to avoid changing the predefined hopset. Older AFH schemes exist primarily in the HF bands, but newer AFH systems operate in the widely-used ISM

and UNII bands and are focused on hopset collision avoidance. This research also avoids hopset collisions. Additionally, the proposed system uses a combination of random keying and spectrum sensing to accurately avoid occupied frequencies using frequency hopping.

2.3 RF Spectrum Sensing & Mapping

2.3.1 Cooperative Sensing.

Cooperative sensing consists of multiple CRs within a CRN working together to formulate an accurate picture of the RF environment. This topic is of great interest in distributed CR systems research as cooperative sensing both minimizes overall workload for the individual node and yields a more reliable depiction of environmental conditions. However, there are tradeoffs to cooperative sensing: the CRN must both communicate valuable RF spectrum information within the network and minimize the amount of network flow over low-bandwidth and potentially vulnerable control channels.

In [26], the authors examine the physical aspects of the motivation for cooperative sensing. The authors recognize bandwidth as a confounding factor of the amount of information detectable by CRs: if a radio has a low-bandwidth control channel, it is likely only able to perform either energy detection or signal statistics; a CR with a high-bandwidth control channel, however, may implement all possible detectors. The type of control channel is largely dependent on the hardware used to implement the board.

In line with opinions presented in [26], the authors of [27] also examine the issue of bandwidth limitations in cooperative sensing. Nodes must report within the network their observations, and the granularity of these observations is limited by the bandwidth delivered in hardware. The implementation studied in [27] uses sensing bits to quantize a likelihood ratio (LR) for determining which nodes have enough worthwhile information to disseminate through the network. Using LRs is thereby a means for reducing meaningless observation traffic within the network via low-bandwidth control channels. Their tests show

that with only a minor cost to sensing performance, the number of sensing bits decreases greatly.

Cooperative sensing is also very applicable to cluster-based networks, as noted in [28]. Sensing in this manner minimizes overall traffic throughout the overall CRN and allows for sensing to be geographically localized. This is important because geographically distant nodes are unlikely to sense and attempt to use the same RF spectrum environment as nodes within close proximity of one another.

Cooperative sensing plays a large part in this network in terms of data fusion (RF map merging), and in order to minimize overhead time associated with adapting to the RF environment, limiting extraneous information over control channels is vital. This research proposes minimizing network traffic not via minimal data representation as in [27], but instead by partitioning the network in to sub-networks ("subnets") using clustering. There are two distinct benefits to partitioning the network into clusters that are likely to operate in the same RF environment.

2.3.2 Spectrum Map Storage & Usage.

An RF spectrum can be represented in many different ways, such as `real+complex` number pairs, a histogram-like FFT model, and so forth. Prior to conducting this experiment, it is necessary to choose a method such that it is both portable between components and extensible for future research. One method generalizes the RF spectrum as part of a larger set of data, the radio environment. In their pioneering Radio Environment Map (REM) research, Zhao, Le, and Reed present the RF spectrum as part of a larger model containing multiple quantitative and qualitative radio environment characteristics to include terrain, historical or time-based usage data, and others in addition to the obvious inclusion of present RF spectrum usage [10].

This research implements the REM solely as an avenue for storing spectrum availability data where frequency-specific data is contained in a binary vector. This concept

is validated in section three. Therefore, and for the rest of this paper, the spectrum map is referred to simply as the REM to allow for expansion of this research to include more aspects of the radio environment.

2.3.2.1 Radio Environment Map.

Zhao, Le, and Reed introduce the radio environment map (REM) as an integrated database consisting of multi-domain information such as geographical features, available services, spectral regulations, locations and activities of radios, policies of the user and/or service provider, and past experience." Figure 2.5 illustrates this concept.

A cognitive radio (CR) uses this REM to improve its situational awareness (SA) and reasoning process, and it disseminates/receives REM information to/from other CRs in order to form a highly situation-aware CR network. According to the authors, the formation of a REM is both a system-level solution to cognitive networking" and a natural step from legacy radios to CR; more succinctly, REM usage in CR networks leverages prior knowledge and collective intelligence." Finally, the authors of [10] recognize five open questions related to the REM, two of which apply specifically to this thesis:

1. How can we assure the REM's integrity, security, privacy, and reliability?

2. In order to provide desired performance, how current and what level of granularity does the information contained in the REMs need to be?

Regarding these questions, this thesis concerns the REM dissemination integrity, REM reliability, and REM currency.

2.3.2.2 Overhead Analysis for REM-enabled Cognitive Radio Networks.

An important aspect of using a REM in conjunction with a CR is the overhead associated with such action. Zhao, Reed, Mao, and Bae [29] present REM dissemination schemes and issues, analyze REM dissemination overhead, and compare various overhead scenarios with network simulation results. This research mainly pertains to REM

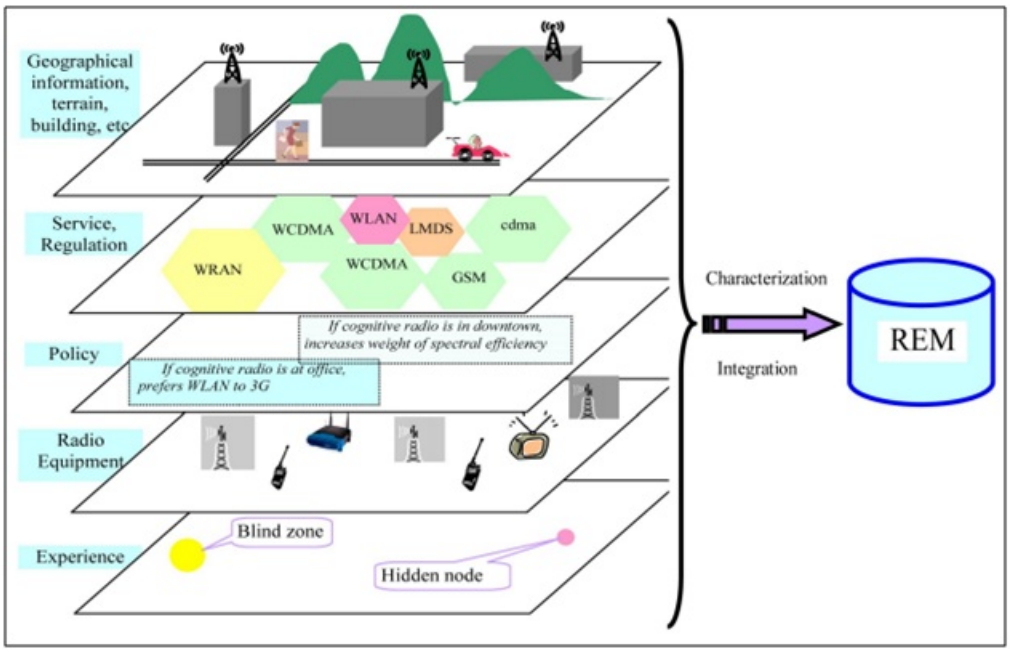


Figure 2.5: Possible REM characteristics [29].

dissemination schemes and issues with an eventual focus on simulating the network and analyzing subsequent REM overhead.

Dissemination schemes include periodic REM broadcasts to the entire network (via plain flooding), an CR-specific extension of the mobile wireless LAN optimized link state routing (OLSR) protocol (via adapting or extending the OLSR HELLO or Topology Control (TC) messages), adapting the REM dissemination rate based on existing primary users (PUs) or interference in an application-specific ad hoc network, and disseminating REM information in an on-demand" scheme (i.e. only transmitting requested REM information, as opposed to always transmitting the entire map) [29].

When choosing the REM dissemination scheme, the overarching goals are minimal retransmissions, minimal transmission sizes, and a minimal number of REM sources. REM implementations can be any data structure containing the desired information. In ascending computational overhead, the implementation can take the form of a multi-dimensional

matrix, a C++ structure, or a multi-dimensional database. REM memory footprint depends on both the implementation and the amount of information stored in the REM. The authors suggest using a common control channel in any of the following flavors: Narrowband channel in a licensed band; channel in license-free ISM or UNII band; an ultra-wideband (UWB) channel; or sharing with the traffic channel [29].

While the aforementioned methods are valid, this research proposes REM dissemination by a different means. *Totally-ordered multicast* (TOMC) is an information dissemination protocol in which all nodes receive messages in the same order [30]. Such a protocol is valuable in a network where information must be distributed among a large number of nodes and network structure decisions are contingent upon that information. This is essentially the state of the proposed network during formation and prior to clustering. Nodes must receive information in the same order to guarantee that each node receives the same data in the same order. While TOMC is not explored further than conceptual integration in this document, it is expected to become a vital part of future research involving inter-node communication.

Table 2.5 describes a potential implementation for REM information, ideally contained in a C++ structure or multi-dimensional database. The rightmost column shows how each field applies to the system proposed in this research, if applicable. While the IEEE 802.22 standard is designed to be infrastructure-based, our CRN is is designed as an infrastructure-less, *ad hoc* network. This table helps define where the proposed system is similar to an infrastructure-based CRN like one governed by IEEE 802.22 [10].

Table 2.5: REM information element for IEEE 802.22 systems [10].

Domain and index range	**Syntax and index**	**Application to research**
Application type (700-799)	Voice (701), packet data (702), vido conference (703), etc.	Packet data
Optimization layer (600-699)	Minimize interference to PU (600), maximize SU throughput (601), etc.	Maximize SU goodput
Topology and network type (500-599)	Infrastructure-based network {WCDMA (500), cdma2000 (501), WRAN (502), etc.}; *Ad hoc* network (510), mesh network (520), etc.	*ad hoc* network
MAC and duplex (400-499)	TDMA (400), FDMA (401), CDMA (402), OFDMA (403); FDD (410), TDD (411), etc.	Frequency hopping
Geography and mobility information (300-399)	Indoor {home (300), office (301), airport (302), factory (303), etc}; outdoor {urban (310), suburban (311), open rural (312), highway (313), etc.}; in-vehicle {train (320), bus (321), car(322), plane (323), etc.}; etc.	Various
Modulation type (200-299)	AM (200); FM (210); M-PSK {BPSK (220), QPSK (221), etc}; M-QAM {16-QAM (230), 64-QAM (231), etc.}; etc.	N/A
Radio device capability (100-199)	Channel coding {Convolutional coding (100)}, Turbo coding (110), etc.}; maximum RF transmit power (120), sensitivity (130), operational bands (140); antenna type (150), etc.	Sensitivity, operational frequencies
Experience (0-99)	Blind zone (10), hot spot (20), hidden node (30), etc.	Dynamic RF environment

2.3.2.3 Radio Environment Map-enabled Learning Algorithms.

The authors of [31] propose a generic top-down approach for obtaining situational awareness (SA) via REM exploitation for CRs and a framework for CR learning loops. Additionally, the authors conduct simulations to evaluate the efficiency and effectiveness of the approach and framework they propose. Metrics include adaptation time, average received signal to interference and noise ratio (SINR), average throughput, and existing PU average packet delay.

The authors use Table 2.5 to describe one possible method for digitizing and indexing radio environment information. For example, suppose that the REM shows that the CR is within the service area of a wireless regional area network (WRAN), and that the REM also lists TV channel 9 as available for use. Once the CR senses the open channel and adjacent channels and confirms that TV channel 9 is indeed available and suitable for use, the WRAN determines the current situation, which in this example can be defined by the Table 2.5 as {703, 600, 502, 403, 411, 312, 231, etc.}. Thus different REM/CR scenarios can be modeled across different CRs [29, 31].

The use of the REM has also been explored as a means to spatially interpolate radio locations. The authors of [32] present a new robust method for using inverse distance weighting to determine feature locations. The REM forms the backbone of several other areas of research, including building a mobile *ad hoc* network architecture around the REM [33] and investigating the effect of a noise threshold on REM effectiveness [34]. Both concepts are central to this research–the REM is the primary means for storing, using, and transferring this system's RF environment observations, and the applicability of a noise threshold is also thoroughly vetted when comparing the RF spectrum environments of neighboring CRs.

2.3.3 Threshold Detection.

Threshold detection is an important part of extracting valuable information from a signal. While developing a threshold detection method is not a focus of this research, implementing an existing solution plays a major role in both the clustering evaluation and hopset selection steps of the overall cycle of operations. The authors of [35] highlight one such method using the noise floor. The noise floor is determined to be the mean of the entire spectral density function, and the threshold is the noise floor (mean) multiplied by some coefficient. This coefficient serves to move the threshold above the noise floor by some amount. In this research, the threshold is implemented in a similar manner for the purpose of converting the measured spectrum to a format usable for comparison and lightweight hardware manipulation. The conversion method and the usage of the conversion output is discussed in Section 3.2.5.1 and validated both Sections 3.2.5.3 and 3.3.4.2.

2.3.4 Summary.

In order for a radio to accurately sense its environment and correctly adapt its behavior, it must be able to "see" its surroundings and "remember" how they affect its operation. Many different techniques for sensing in the context of CRNs currently exist and have been demonstrated as effective purveyors of RF environment information. The radio environment map (REM) provides an ideal medium for efficiently storing and communicating RF environment information. While this research does not implement a spectrum sensing implementation, a method for mapping the spectrum and storing it in a custom REM is proposed and demonstrated.

2.4 Clustering

2.4.1 Overview.

Clustering is the manner in which similar objects are "clustered" together using some distance or similarity metric. In practice, clustering serves a number of purposes. In statistical analysis, clustering is a way to identify groupings of data. In image processing,

clustering enables similarly-colored pixels to be marked as such. This research uses clustering to form IP subnets from an initially-flat radio network.

2.4.2 k-means Clustering.

As Kanungo, *et al* explain in [36], although there are a variety of *k*-means clustering implementations, Lloyd's algorithm is known effectively as the defacto *k-means algorithm*. Before defining the algorithm, though, it is necessary to define terms. First, a *centroid* is simply the geometric weighted center of a cluster. For example, if all nodes in a cluster are of equal weight, the centroid is the center of mass of the shape composed by the nodes. Second, a center is the point location of the centroid. After the clustering algorithm runs, every node is assigned to a center. Finally, a center's neighborhood is defined as those nodes for which z is the closest neighbor. The focal point of Lloyd's algorithm is observing the optimal placement of the center of at the centroid of the relevant cluster. The actual algorithm is defined below.

```
1: procedure LLOYD(Some set of k centers Z)
2:     repeat
3:         for each center z ∈ Z do
4:             Let V(z) denote z's neighborhood
5:             Move z to the centroid of V(z)
6:             Update V(z) with the distance from each point to its nearest center
7:         end for
8:     until some convergence condition is met
9: end procedure
```

2.4.3 University of Maryland Testbed.

The authors of [37] present a framework for testing the *k*-means clustering algorithm. This testbed is written in C++ and forms the centerpiece of this work's clustering experiment. A total of 34 different runtime options are available. For simplicity, and to limit the number of experiments, this research varies several and maintains the rest as default values. One of the most important options is the ability to select among four different clustering algorithms, all of which are variations on the core Lloyd's algorithm.

The testbed uses the original Lloyd's algorithm and three more; the additional three algorithms involve some modification or addition(s) to the Lloyd's algorithm. All four algorithms are listed and summarized below. Full explanations are contained in [37].

- **Lloyd's.** Original centroid-based algorithm that runs until convergence of cluster assignments. Initial centers are randomly sampled.
- **Swap.** Maintains a set of candidate centers and swaps between this list and the existing centers.
- **Hybrid.** Complex hybrid algorithm comprised of both Lloyd's and Swap in which a number of Swap iterations are performed followed by several iterations of Lloyd's. The Hybrid algorithm uses an approach akin to simulated annealing to avoid local minima entrapment.
- **EZ-Hybrid.** Simplified Hybrid algorithm. Performs one swap and a number of subsequent iterations of Lloyd's.

2.4.4 Cluster Visualization.

Every node is associated with a particular RF spectrum measurement. These measurements can be compared between two nodes to yield a similarity ratio ranging from totally dissimilar (zero) to totally similar (one). While several cluster visualizations are informally presented in [36], this research shows both cluster assignments via projected communication links and the RF spectrum similarity between nodes.

2.4.5 Summary.

k-means clustering is the basis of the middleware's ability to reduce a flat network into a hierarchical set of sub-networks. In the context of this research, an IP-based network composed of cognitive radios is partitioned into subnets. While the *k*-means algorithm itself is not new, implementing the algorithm in the realm of congitive radio networks and the manner in which experimental data is visualized is a novel contribution.

2.5 FPGA-Based Cognitive Radio

2.5.1 Kansas University Agile Radio (KUAR).

The University of Kansas uses a Virtex-II Pro FPGA for the digital signal processing operations and digital communication components of their software-defined radio platform KUAR. It performs communications processing in VHDL on the FPGA, does signal processing, radio control, and RF environment sensing with both VHDL on the FPGA and C code on the FPGAs embedded processor, and interfaces with a transceiver and a Linux control processor [38].

2.5.2 Wireless Open-Access Research Platform (WARP).

Rice University uses the PowerPC processor on the Virtex-II Pro FPGA for communication processing on their Wireless Open-Access Research Platform. WARP is used to prototype wireless networks. They have tested and verified the FPGAs on the WARP board in over-the-air tests. The single-input single-output (SISO) Orthogonal Frequency Division Multiplexing (OFDM) transceiver uses the FPGA for all the baseband processing. It uses the radio daughtercards on the WARP board to convert to the RF band [39].

2.5.3 Trinity College's Cognitive Radio Framework.

The authors of [40] at Trinity College performed a case study that demonstrated the implementation of cognitive radio on an FPGA. They created a set of tools for radio designers to be able to implement cognitive radio on FPGAs. This research used a partial reconfiguration of the FPGAs during runtime, allowing a cognitive engine to reconfigure both the software and hardware.

2.5.4 Berkeley Emulation Engine 2 (BEE2).

The BEE2 is a modular radio prototyping testbed for examining both narrowband and wideband approaches, and uses a combination of Simulink and Linux alongside the BORPH operating system in implementation. In their experimental design, the authors of

[14] use the BEE2 as the basis for two separate examples, a wideband configurable testbed in the 400-500 MHz range, and a narrowband multiple-input/multiple-output CR example in the 20 MHz range. This example uses a Virtex II Pro FPGA as the central part of the BEE2's CR extension, much like the implementation proposed in this research.

2.5.5 Virginia Tech Public Safety Cognitive Radio (PSCR).

The authors of [15] present a CR implementation specifically for public safety (police, fire, etc.) personnel, Public Safety Cognitive Radio (PSCR). The research implements the traditional operation cycle of a cognitive radio in very fine detail. Included in the implementation is the radio environment map (REM) much like in this document's proposed system. In addition to the radio's "cognitive cycle" (as also defined with this research), the authors present a fully-described product from physical-layer hardware up through the graphical user interface (GUI). One important feature of the PSCR is its dependency on policy-based verification such that the radio fits both the physical RF spectrum environment and the public policies governing spectrum usage [15].

2.5.6 Summary.

Like those in previous work, this research leverages the FPGA features of flexibility and compact area. In using the base WARP system, a custom IP core written in VHDL, and the Virtex IV's embedded PowerPC core with C code to implement the radio and hopset selector.

2.6 Background Summary

Many underlying technologies compose what we envision to be a viable, field-deployable cognitive radio network and the supporting architecture. While some existing systems are indeed similar to the one proposed in this research, this work employs a wholly new architecture and implements such components in a new and novel manner. Dynamic adaptive frequency hopping, network clustering, and packaging the system into a hybrid hardware/software FPGA are the core pieces of this work and have already been explored

or accomplished individually. This research shows that such sub-systems can be integrated into a novel cognitive radio middleware architecture as part of an ongoing effort to develop a “smarter radio.”

3 Methodology

This research proposes the architecture for a frequency hopping cognitive radio network which coexists with other RF spectrum users, and implements several of the necessary features. Specifically, the goals of this research are to answer the following questions:

1. Is it possible and feasible to implement such a system?
2. What are the properties of such a system?
3. What are the pieces for implementing such a system?
4. How is the system implemented?
5. What is the operation of the system?

Similarly, the following hypotheses are investigated:

1. *Feasibility of clustering an ad hoc CR network based on spectrum similarity and physical distance.* The network is partitioned, or clustered, according to the similarities between spectra at individual nodes. It is expected clustering a CR network according to local RF spectrum similarity and physical distance produces clusters with nodes whose sensed spectra are more similar than clustering on physical distance alone.
2. *Feasibility of AFH in an interference environment.* Since the core focus of this system is to adaptively select and use a frequency hopping sequence, or *hopset*, based on the existing RF spectrum, it entails that this research must prove such operation as tractable. It is expected the system can recognize available and unavailable spectrum, react with an appropriate AFH sequence, and continue normal operation.

3. *Feasibility of communicating information between nodes in an orderly, organized fashion.* Nodes must have the ability to form and maintain a dynamic network, share spectrum data, and achieve synchronized frequency hopping activity. Since multiple nodes must communicate on the same hopping pattern, they must chronologically share spectrum resources. Further, the spectrum may change independently at different network locations. To maintain adaptive operations, nodes must communicate spectrum data, engage their cognitive operations, and (re)distribute an adaptive hopset. It is expected all nodes can maintain effective communications while conducting AFH operations.

4. *Ability to successfully transmit in the presence of a dynamic RF spectrum.* In addition to deciding an adaptive hopset, the system must transmit and receive between radios. This is necessary to measure typical traffic-based network metrics such as goodput and latency. It is expected the system can combine the first three hypotheses to form a novel AFH CR network system.

3.1 Whole System

This section presents the entire adaptive frequency hopping (AFH) architecture. When in use, the system is able to coexist with other spectrum users. This section explains how the system functions and its internal structure and lists which components are implemented in this research.

3.1.1 Assumptions.

In order to properly scope this research and therefore limit the design decisions for unimplemented components, the following assumptions are made:

1. Rendezvous and routing are complete prior to middleware actions.

2. Every radio is within transmission range of all other radios.

3. Every radio acts as a node in an IP network such that each radio has both an IP and MAC address.

4. Radios have the ability to communicate with one another prior to network clustering.

5. The network is immobile and the spectrum is fixed, i.e., nodes cannot enter or leave the network and the RF spectrum remains constant once the experiment begins.

6. Nodes are uniquely identified based on network arrival time, i.e., the first radio to join the network has the first identifier.

7. The radio with the first identifier (e.g. "Node 0") serves as the network leader.

8. Each radio can detect "quiet" periods and engage in spectrum sensing during these periods.

9. Nodes have the ability to synchronize their time upon rendezvous and remain synchronized for the duration of operations.

10. The observable spectrum is contiguous and spans the same frequency range across the network such that all REMs are of equivalent structure during map merging.

11. Operations occur precisely in the order defined in Figure 1.4 such that transmission and sensing cannot occur simultaneously.

12. A reliable transfer protocol is used during data transmission in **O-3** and **O-6** in Figure 1.4.

3.1.2 Whole System Function.

System functionality describes the "what" of the proposed architecture. In other words, the system function is an direct answer to the requirements posed at the beginning of this chapter and accounts for the assumptions given above. The following list details

each operation in chronological order, and Figure 3.1 shows how these operations relate to each other and the conditions that must be met to transition between operations.

O-1. Sense RF spectrum. All nodes scan the radio frequency environment to obtain a snapshot of occupied frequency bands and available whitespaces. Effectively, this snapshot amounts to a 2,048-bin Fast Fourier Transform (FFT) of the analog spectrum. A threshold is applied to the FFT output to convert the spectrum to a REM representation.

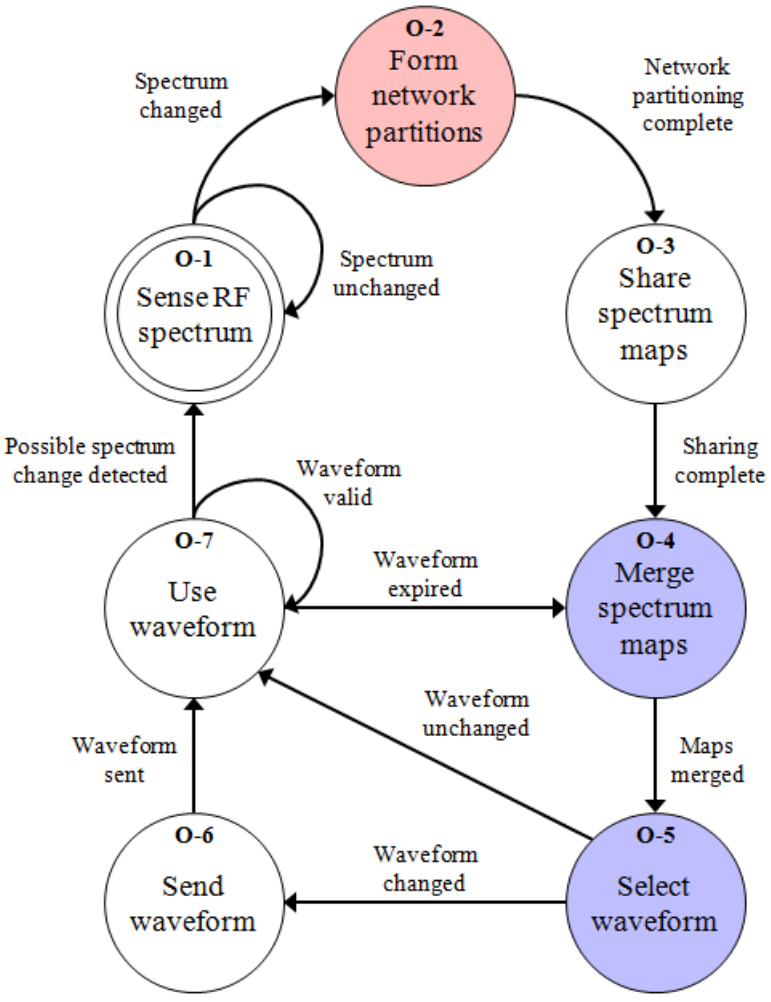


Figure 3.1: Whole system function.

Each node applies this threshold individually and stores this information into its local REM. This threshold is discussed in Section 3.2.5.1.

O-2. Form clusters. During ad-hoc network formation, each node must perform rendezvous/neighbor discovery. Per assumption one above, this is already accomplished prior to **O-1**. For the purpose of this research, it is assumed nodes begin in a non-hopping mode of operation, where discovery occurs on a fixed control channel. Per assumptions two and three above, the radios initially form one large, flat-hierarchy IP network. Nodes then form geographic clusters, meaning nodes within close geographic proximity are assigned a common cluster identifier. These clusters serve as IP subnets, and each cluster/subnet has a leader given by assumption six. Such subnets serve to reduce the total amount of traffic within the overarching network. It is hypothesized nodes within a cluster have similar REMs and are able to form a common REM with limited data exchange.

O-3. Share REMs. A radio may freely transmit their REM information at any time, but will only do so if its observed RF spectrum environment has changed. This is accomplished via TOMC. In TOMC, many nodes can attempt to "speak" at a given time, but all nodes will only "hear" from one node at a time due to the nature of the protocol. In this protocol, all intended transmissions are eventually received in a *totally-ordered* fashion, meaning each node will receive all data in the same order as all other nodes. This operation occurs after **O-2** because doing so in a flat, complex network floods the network with REM data traffic, thereby voiding the concept of using clustering to minimize unnecessary transmissions.

O-4. Merge spectrum maps. The leader of a given cluster merges the maps of the cluster's nodes to form a common REM. Because REMs are represented as binary data, the common REM is simply the result of a logical `AND` operation between all REMs. This operation is performed within a circuit on an FPGA. Section 3.3.4.2 validates this concept.

O-5. Select hopset. Using a random key, an FPGA circuit randomly selects available channels for use as a hopset. This hopset is 2,048 hops in length. Section 3.3.4.3 validates this concept.

O-6. Send hopset. As in operation **O-3**, the radios use TOMC to distribute an adaptive hopset through the network. Hopsets are only valid within a subnet, i.e., each subnet generates and uses a unique hopset.

O-7. Use hopset. Once all nodes within a cluster have knowledge of the adaptive hopset, the radios commence data transmission and reception. This operation is the result of the preceding six, and functions just as any typical frequency hopping radio.

The system returns from ***O-7*** to ***O-1*** when one of two conditions are met: (1) the hopset expires, or (2) the spectrum has potentially changed. A hopset expires if all hops have been used. While individual frequencies may be reused in order to generate a hopset of desired length, the hopset itself is generally for one-time use only. However, if for any number of reasons (i.e., present environment is deemed benign, a new key is unavailable, etc.) a new hopset cannot be generated, the hopset is reusable. The spectrum is known to have potentially changed if any radio sends new REM data during any operation other than ***O-3***.

Of the seven operations outlined above, operations ***O-2***, ***O-4***, and ***O-5*** are implemented in this research. These operations are shaded green in Figure 3.1. Operations ***O-1***, ***O-3***, ***O-6***, and ***O-7*** are only explained in this section and are not validated any further in this work.

3.1.3 Whole System Structure.

The solution proposed in this research is a framework for accomplishing the operations described above. Specifically, a middleware architecture for adaptive frequency hopping within cognitive radio network is presented. This architecture exists in a hybrid hardware-software system on a WARP II FPGA-centric radio board. This middleware architecture is shown in Figure 3.2.

In Figure 3.2, there are four primary components: the RF spectrum emulator, a database for storing spectrum map data, the radio hardware with which the radio transmits and receives, and the middleware architecture itself. The architecture itself consists of eight sub-components: database interface, spectrum map parser, map distributor, clustering, map merger/hopset selector, secure hash chainer, multicast communication layer, and hardware interface layer. Of these sub-components, the clustering and map merger/hopset selector are constructed and tested for this work (shown here in green). The remaining pieces (i.e., on-board spectrum sensing, radio card operation, and network communication) are to be demonstrated in future iterations of this project. The overall system and its operation is proposed in Section A.

The clustering implementation is integrated into the architecture purely in software. In order to minimize traffic within the overall network, the radios exist within a packet-switched network. Geographically-close nodes belong to the same subnet. By using a clustering algorithm to partition a collection of many nodes, it is expected nodes contained within the same subnet will have highly-similar RF spectra. In Section B, an experiment is presented for selecting the best combination of number of clusters (equivalent to number of subnets) coupled with the ideal clustering algorithm heuristic.

The map merger/hopset selector component exists primarily in hardware with limited software interfacing. After accepting an indefinite number of maps from radios within the network and a random key, it generates an adaptive hopset which maps to the frequencies (or channels) usable by radios within the network. This capability allows the system to fit within the available spectrum "whitespace" in true DSA fashion. This design is validated in Section C.

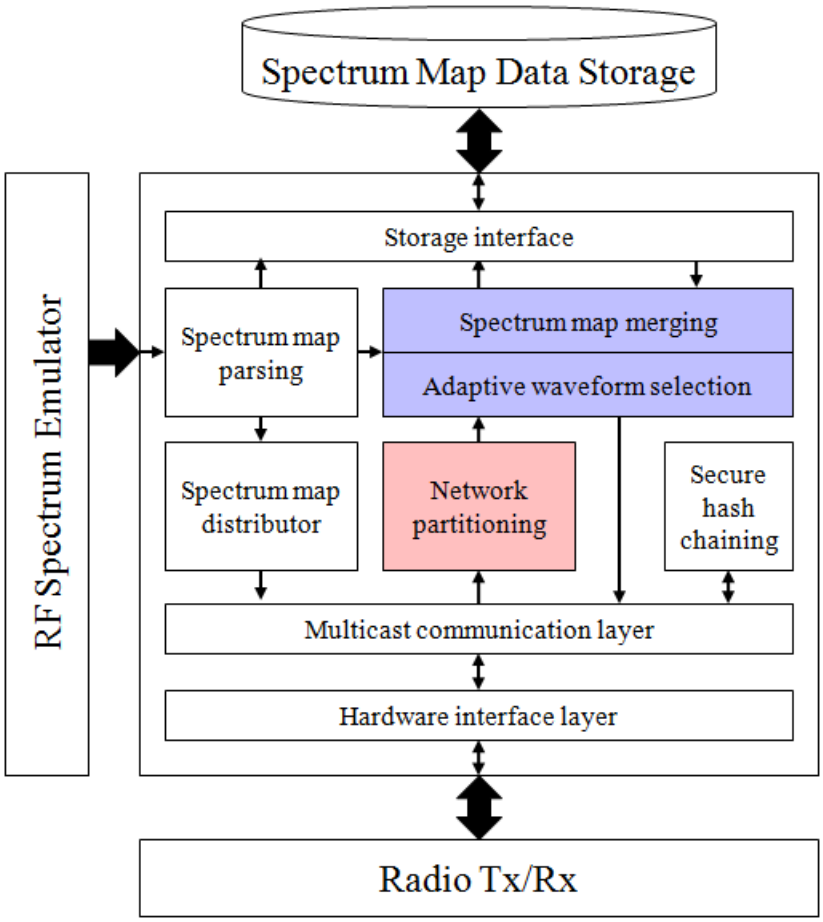


Figure 3.2: Proposed middleware architecture.

3.2 Network Clustering

3.2.1 Problem Definition.

3.2.1.1 Goals and Hypothesis.

The goal of this experiment is to demonstrate a method for selecting a clustering algorithm for integration with the rest of the middleware architecture. The "defacto" k-means algorithm is Lloyd's algorithm [36, 37], so Lloyd's algorithm is the default choice. Therefore, the purpose of this experiment is to investigate which clustering heuristic described in Section 2.4.3, if any, performs best over a range of RF spectrum environments and center counts.

3.2.1.2 Approach.

We introduce a method by which the four clustering heuristics in Section 2.4.3 can be evaluated for expected best performance in a cognitive radio network. We first establish a baseline for clustering performance using "canned" maps and then use the point distribution generator within the `kmltest.exe` software to form "real" maps of four distribution types: uniform, Gauss, and multi-cluster. The points used for these maps represent receivers in a network, as opposed to the transmitters we intend to avoid.

Apply threshold. In the actual system, operation **O-1** (see Figure 3.1) converts the observed spectrum to a format usable by the map merging and hopset selection component described in Section 3.3. This conversion is also necessary to validate the clustering test framework proposed in this section. Because the spectrum is input as an unprocessed, emulated entity for this experiment, the threshold conversion is completed within the test framework.

Establish baseline. In order to validate the performance of `kmltest.exe`, we form several trivial maps to verify that the four clustering heuristics perform as expected. For example, if there are n discrete clusters as seen by the human eye, it is expected n clusters

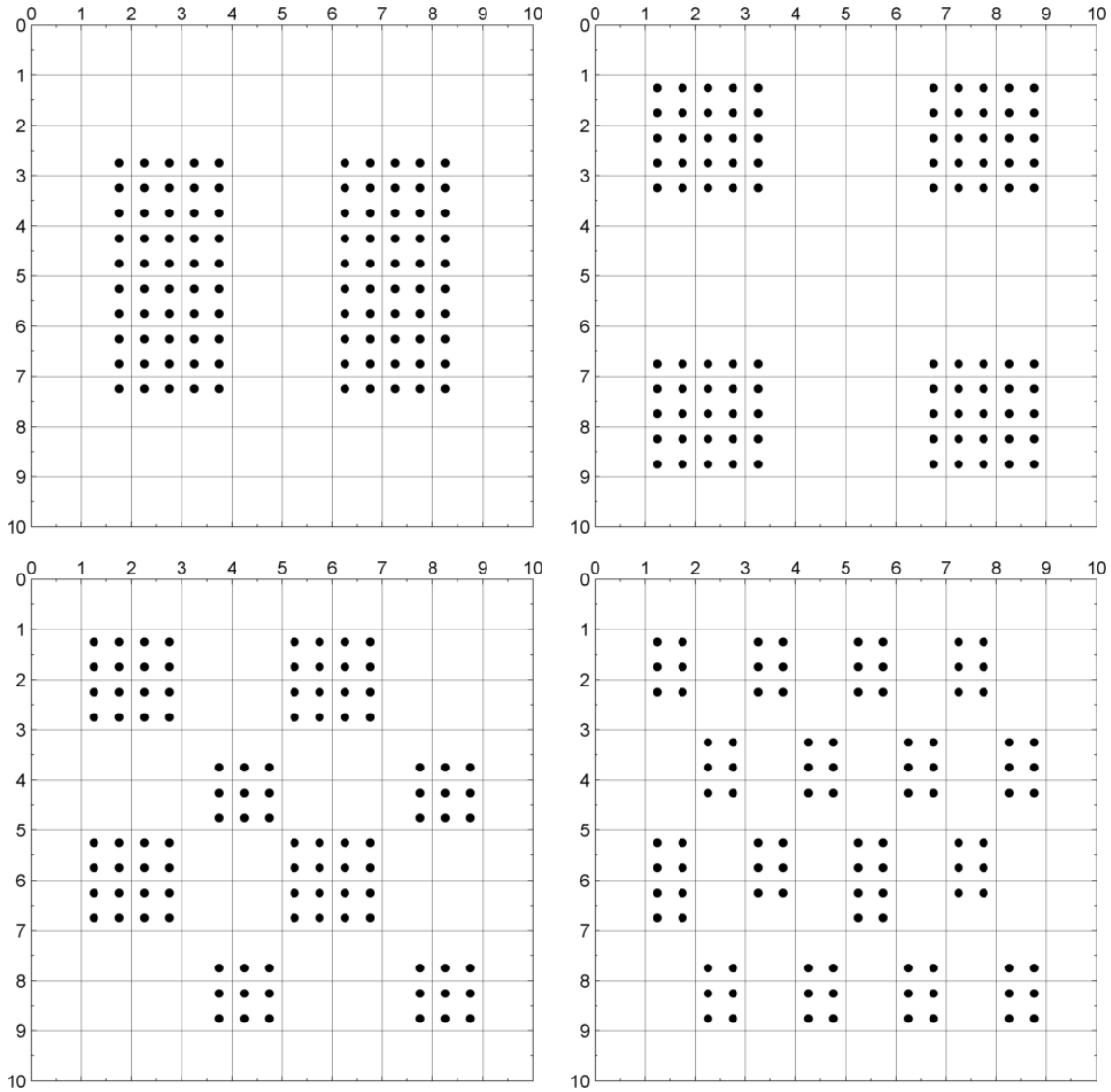


Figure 3.3: Baseline receiver configurations.

will be formed as they are geographically positioned. The baseline receiver configurations (i.e., the canned maps) are shown in Figure 3.3.

Clustering algorithm testing. The clustering algorithm partitions a flat cognitive radio network into sub-networks (analogous to IP subnets). This partitioning decomposes from a larger problem in which every node can "talk" to every other node (notionally

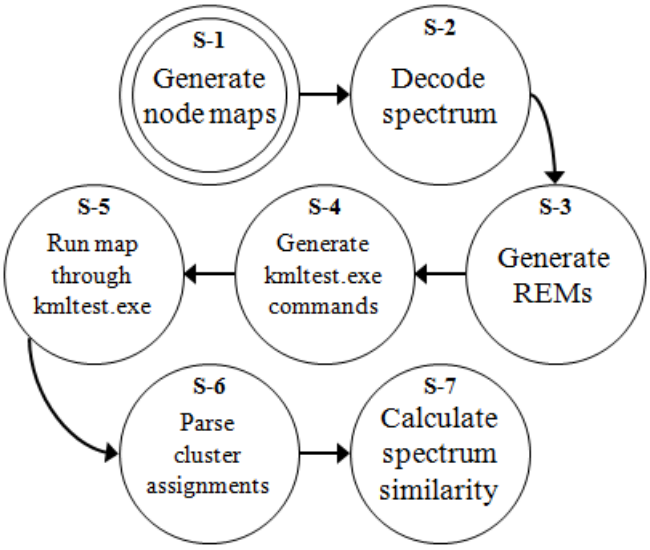


Figure 3.4: Network clustering test framework function.

of $O(n^2)$ complexity) into a simpler problem (notionally toward $O(n\ log(n))$ complexity). (Note: Network complexity is not proven as part of this research, and is used in this research as a method for generalizing expected trends.) The most desirable performance yields the highest spectrum similarity within a cluster such that extraneous traffic is minimized and the adaptive hopset is chosen from the broadest possible space.

Results evaluation. Using the data from this experiment, we implement one or more heuristics for use in the overall cognitive radio network. Selection is based on best performance across a range of RF spectrum environments and node configurations.

3.2.2 System Services.

S-1. Generate node maps. Given the number of nodes, the number of maps needed, and map dimensions, component ***C-1*** generates the desired number of maps with the desired quantity and size using the specified distribution. There are no bounds on the inputs. The generated maps can be reused in future tests for a consistent dataset.

S-2. Decode spectrum. Spectrum data is originally formatted according to the virtual device that recorded the measurement. Given the original data and the map dimensions, component ***C-2*** extracts that data to produce a grid representation of the RF spectrum of the same dimensions as the maps generated during ***S-1***. The output of this component takes the form of multiple discrete FFTs.

S-3. Generate REMs. Using the data generated by ***S-2***, component ***C-3*** transforms the spectrum data into binary REMs. These REMs are used for calculation of the performance metric.

S-4. Generate `kmltest.exe` commands. Clustering is performed in an external program, `kmltest.exe`, which implements the KMlocal testbed described in [37]. Normally, the clustering heuristic used and the number of centers are input by the user at command line. To automate the testing process, commands must be prefabricated and piped in using system calls. This service generates a text file containing all commands needed to run the appropriate test.

S-5. Run map through `kmltest.exe`. This service is the core of the experimental design, and is therefore the Component Under Test (**CUT**). The program is run using system calls and input is given via the text file generated during ***S-4***. The program generates output to a predefined text file.

S-6. Parse cluster assignments. When `kmltest.exe` runs, it generates a report containing the assignments of all points. Because this information cannot be returned directly to the test framework without extensive modification, it is more viable simply to parse the output file generated by the *k*-means software. Along with the REMs derived in ***C-3***, cluster assignments are used for calculation of the performance metric.

S-7. Calculate spectrum similarity. The similarity calculation component computes the performance metric described in Section 3.2.5.4. This component accepts the list of all REMs and the list of cluster assignments as inputs and returns a single number.

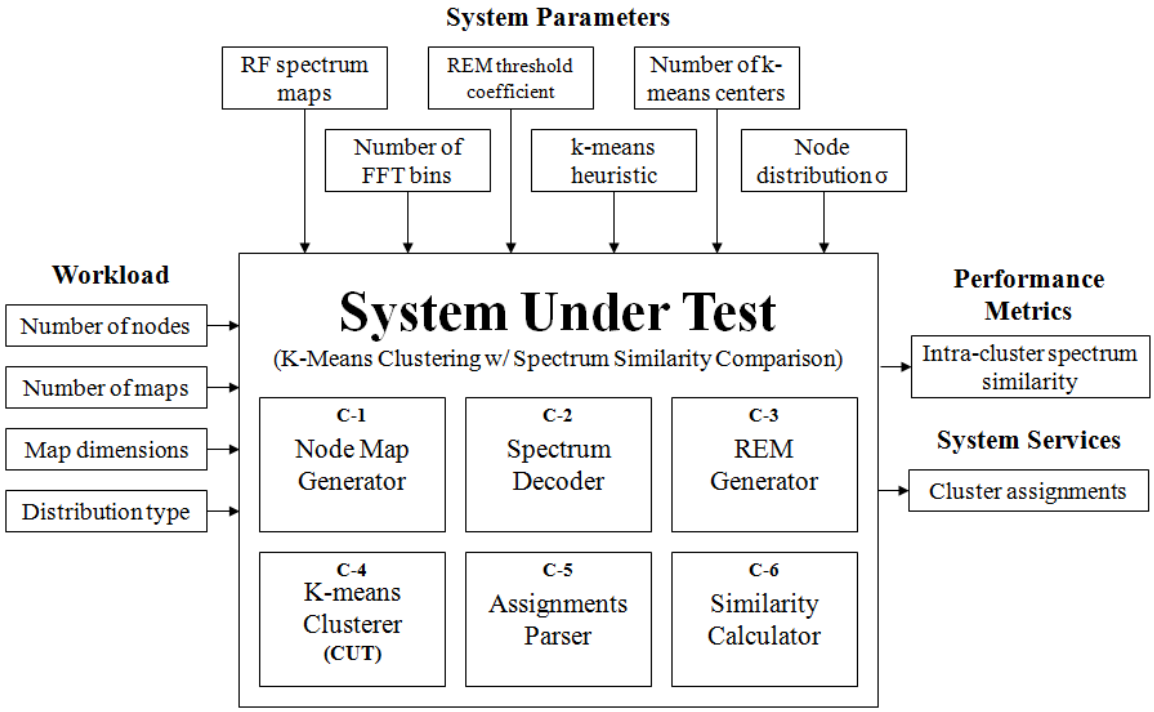


Figure 3.5: Network clustering test framework structure.

3.2.3 System Boundaries.

The system under test (SUT) is comprised of six components:

C-1. Node map generator. This component exists within the *kmltest.exe* program and can generate multidimensional point collections of various distribution types. This research uses two-dimensional maps over uniform, Gaussian, and bimodal Gaussian distributions.

C-2. Spectrum decoder. Data from the is parsed from a structure formed by the DYSE into a structure more easily used in determining spectrum measurements to individual locations.

C-3. REM encoder. Applies the threshold value and forms binary vectors for use in comparing RF spectra.

C-4. k-means clusterer. Performs the clustering operation using one of four heuristics and outputs the results to a file.

C-5. Assignments parser. Reads the cluster assignments from the output file and generates a data structure containing all nodes and their assignments.

C-6. Similarity calculator. Calculates the similarity metric for all clusters.

3.2.4 Workload.

The component under test is affected by four workload parameters: the number of nodes, the number of maps, each map's dimensions, and the distribution type used to create the map. All four parameters are input to the map generator component in the `kmltest.exe` software.

Table 3.1: Workload parameters and descriptions.

Workload Parameter	**Description**
Number of nodes	Defines how many nodes should be created for each map. The number of nodes in a map remains constant for all maps generated during a given test.
Number of maps	Defines the number of maps generated for a test.
Map dimensions	Specifies the x and y dimensions of a map such that the minimal coordinates will be $(1, 1)$ and the maximal coordinates will be (x, y). When nodes are generated, they are placed within these bounds.
Distribution type	Selects the type of distribution used to generate the map of nodes.

3.2.5 Performance Metrics.

This system's performance metric is determined by the similarity between observed spectra at nodes within clusters. This metric is determined using the comparisons of multiple spectrum maps. The method for determining spectrum maps is presented first, followed by an clustering affects a cognitive radio network. Last, a node-wise similarity metric (the main system performance metric) is examined.

3.2.5.1 Spectrum Representation and Threshold Determination.

In this research, the REM is represented as a binary vector. This vector is derived from an analog-to-digital conversion of the analog spectrum. In the case of using a discrete FFT to analyze the spectrum, the number of FFT bins that corresponds to one discrete channel depends on the channel bandwidth required by the user. For example, if there are N FFT bins and M discrete channels span the given frequency band, $\lceil \frac{N}{M} \rceil$ bins map to each channel. We assume that the mapping of bins to channels and/or center frequencies has already been established.

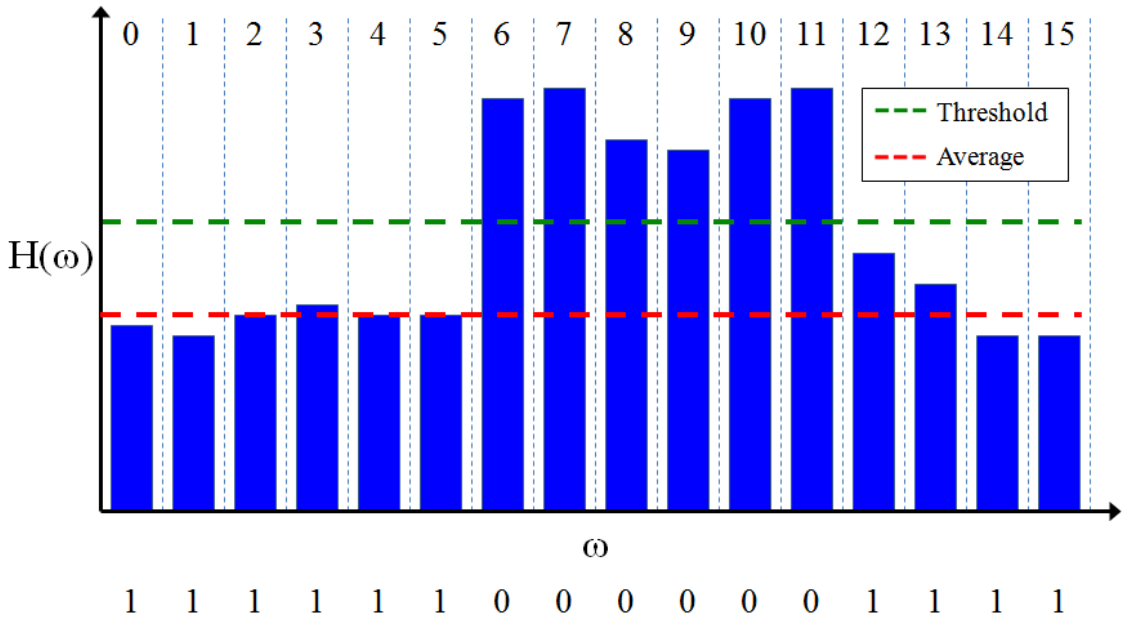


Figure 3.6: Threshold application example.

Using the discrete FFT output, we apply a threshold value to each bin. In [35], researchers present a simple method for applying a threshold to a signal to extract information. The average of each discrete FFT bin's value is multiplied by some coefficient to increase the likelihood that the threshold is applied above the noise floor. In this research, the coefficient is 0.2. This translates to moving the threshold 20% higher than the average FFT bin value when applied to a linear scale (see Figure 3.13). We use this threshold method to evaluate each bin for energy presence. Pilot experiments show that in order to eliminate as much noise as possible without overlooking actual spectrum usage, this value should be held constant at 0.2.

Calculation and application of the threshold is performed in the spectrum sensing operation (**O-1** in Figure 3.1) by marking each bin as a '1' if the spectrum is unused above the threshold. Otherwise the bin is marked as a '0'. The result of this process is the binary vector spectrum representation where ones represent available frequencies and zeros show unavailable spectrum. An example of the threshold conversion method is shown in Figure 3.6, and the threshold is applied to an actual RF map in Figure 3.13.

3.2.5.2 REM Scenarios.

A geographically-separated network does not share a common REM. A single frequency hopping sequence cannot be selected which every node can concurrently use. Individual clusters, however, are more likely to have common unoccupied spectrum. To achieve this goal and build the aforementioned hierarchy, the strategy of clustering is break the network into smaller clusters, which do share common REM. In this manner, Scenario C distills into separate instances of Scenarios A and/or B. As shown in the notional examples of Figures 3.7, 3.8, and 3.9, the greatest amount usable spectrum is available when clustered nodes have similar observed spectra, likely the result of geographical closeness.

S-A, Similar. Nodes are geographically close to each other and have nearly identical REMs. A fused common REM is approximately equivalent to any individual REM within

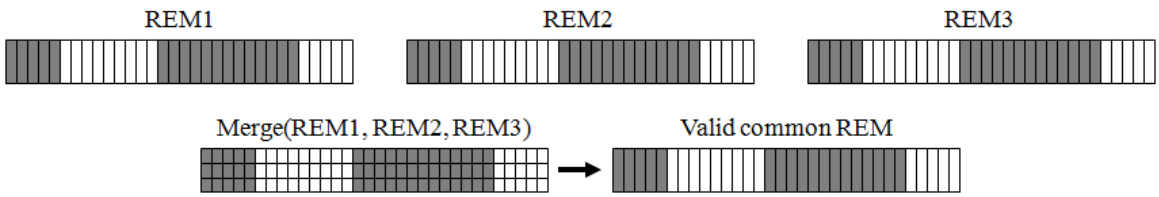


Figure 3.7: Cluster scenario S-A.

the cluster. This is the simplest scenario as the aggregate REM is easiest to compute, i.e., the cluster leader's REM. Any given node can communicate with any other node at will. An example of individual and merged spectra in ***S-A*** is shown in Figure 3.7 where the spectrum is partitioned into 32 bins. Dark areas represent occupied spectrum and light areas represent usable "white space."

S-B, Overlapping. Nodes with moderate geographical separation likely have non-identical, but overlapping, REMs. A fused common REM is equivalent to the intersection of the individual REMs. This scenario is identical to A with the caveat that the space of possible hop destinations is more limited. An example of individual and merged spectra in ***S-B*** is shown in Figure 3.8 where the spectrum is partitioned into 32 bins. Dark areas represent occupied spectrum and light areas represent usable "white space."

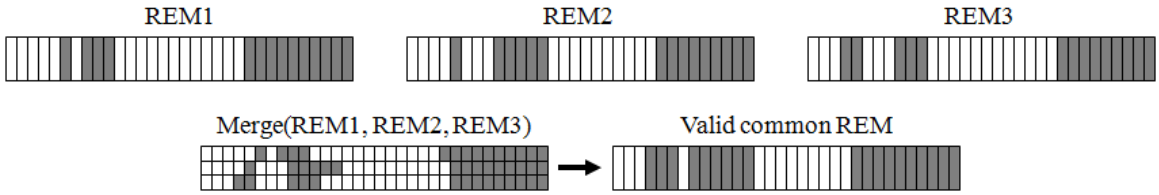


Figure 3.8: Cluster scenario S-B.

S-C, Disjoint. Geographically-far nodes have non-identical and potentially non-overlapping REMs. In this scenario, no more than two nodes share common available

spectrum such that the cluster's REM intersection is disjoint. Therefore, a node has a mutually-exclusive REM intersection with only one other node. In terms of *ad hoc* network communications, this is the worst scenario as a gateway node would need to relay packets between adjacent nodes. An example of individual and merged spectra in ***S-C*** is shown in Figure 3.9 where the spectrum is partitioned into 32 bins. Dark areas represent occupied spectrum and light areas represent usable "white space."

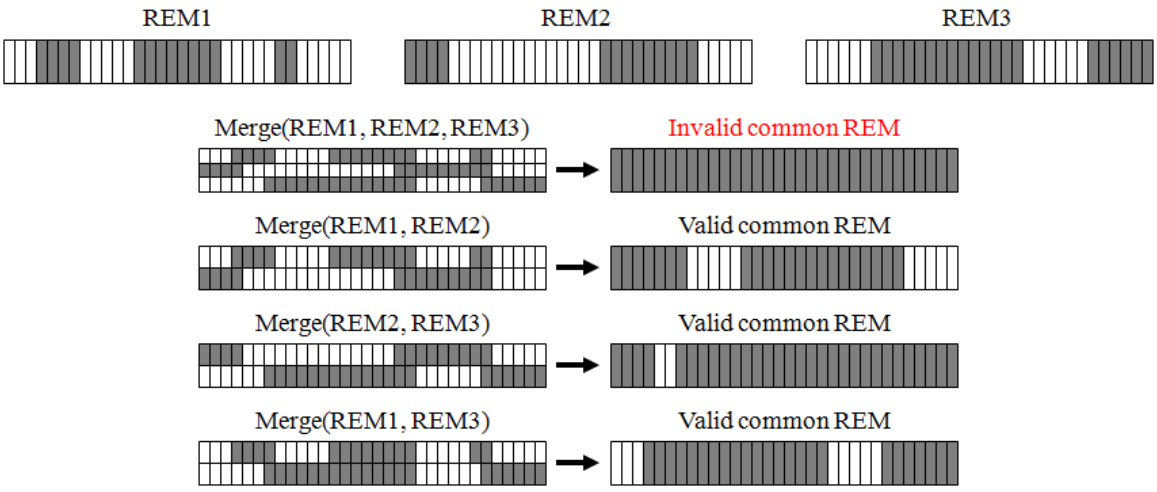


Figure 3.9: Cluster scenario S-C.

Therefore, the goal of clustering is to geographically partition a set of nodes with disjoint REMs (Scenario C) into clusters which have similar or overlapping (Scenario A/B) REMs with the intent of high intra-cluster spectrum similarity. High intra-cluster spectrum similarity implies a broader sample space from which to select an adaptive hopset. A broader sample space, then, is expected to decrease the possibility of interference.

3.2.5.3 Spectrum Map Comparison.

In this research, REMs are represented as binary vectors. As such, it a trivial operation to compare two REMs. Hamming distance, a commonly used metric in coding theory, counts the number of coefficients by which two strings differ. If bits are considered as

coefficients and REMs as strings, the Hamming distance between two REMs of equal length is the number of bits which are different for all equivalent positions. An graphical example of Hamming distance is shown in Figure 3.10. Positions at which bits are equal are shaded green; likewise, unequal bits are shaded red.

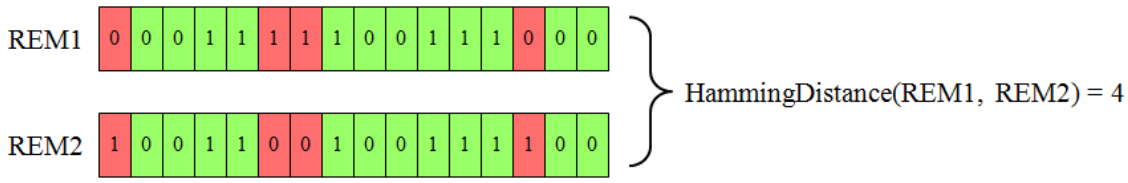
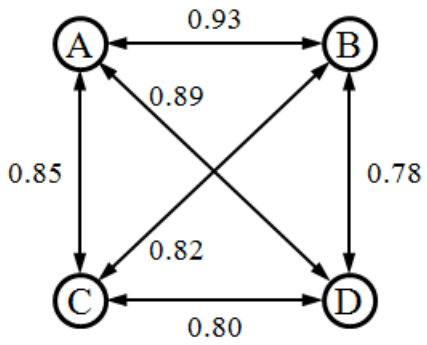


Figure 3.10: Hamming distance example.

Hamming distance is directly applicable to determining the effectiveness of clustering. While the original metric determines the *bit-wise difference* between two REMs, subtracting that number from unity and dividing the difference by the number of bits yields the *percent similarity* between two REMs. The latter is used to evaluate how effectively clustering groups together nodes with like spectra and will be referred to as "REM distance" from this point.

3.2.5.4 Intra-Cluster Spectrum Similarity.

Using the cluster scenarios of Section 3.2.5.2 and the distance metric of Section 3.2.5.3, the test system's sole metric is intra-cluster spectrum similarity (ICSS). To determine the overall spectrum similarity within a cluster, the REM distance is computed for every pair of nodes within the cluster. No pair carries a particular "weight" over any other pair, so intra-cluster spectrum similarity is defined simply as the average of all REM distances within a cluster. An example of ICSS calculation is shown in Figure 3.11 where $ICSS = 84.5\%$.



$$ICSS = (0.93 + 0.89 + 0.85 + 0.82 + 0.80 + 0.78) / 6 = 0.845$$

Figure 3.11: ICSS calculation example.

We do not use Hamming distance as our experimental metric because it is dependent on the number of bits (which map to the bins, channels, frequencies, etc. in the observed spectrum) in the vectors being compared. ICSS is only a slight modification of Hamming distance and, as a ratio, it is independent of vector size. Therefore, ICSS can also be used to compare experiments using arbitrary-length spectrum vectors. (Note: We do not use ICSS for this purpose in this paper, but instead build our experiment for future expansion.)

3.2.6 System Parameters.

The system under test accepts five system parameters: the set of RF spectrum maps, the number of bins used in the FFT (corresponding to the number of bits in a REM), the REM threshold coefficient, *k*-means heuristic, and the number of *k*-means centers. These parameters and the reasons they are chosen are shown in Table 3.2.

The reason(s) for choosing each system parameter follows:

- *RF spectrum maps.* The set of RF spectrum maps is a parameter because varying the RF environment presents different REMs for comparison using ICSS. An RF spectrum map is unique to each grid location and contains the FFT of the spectrum observed at that location. All spectrum map inputs are shown in Appendix A.

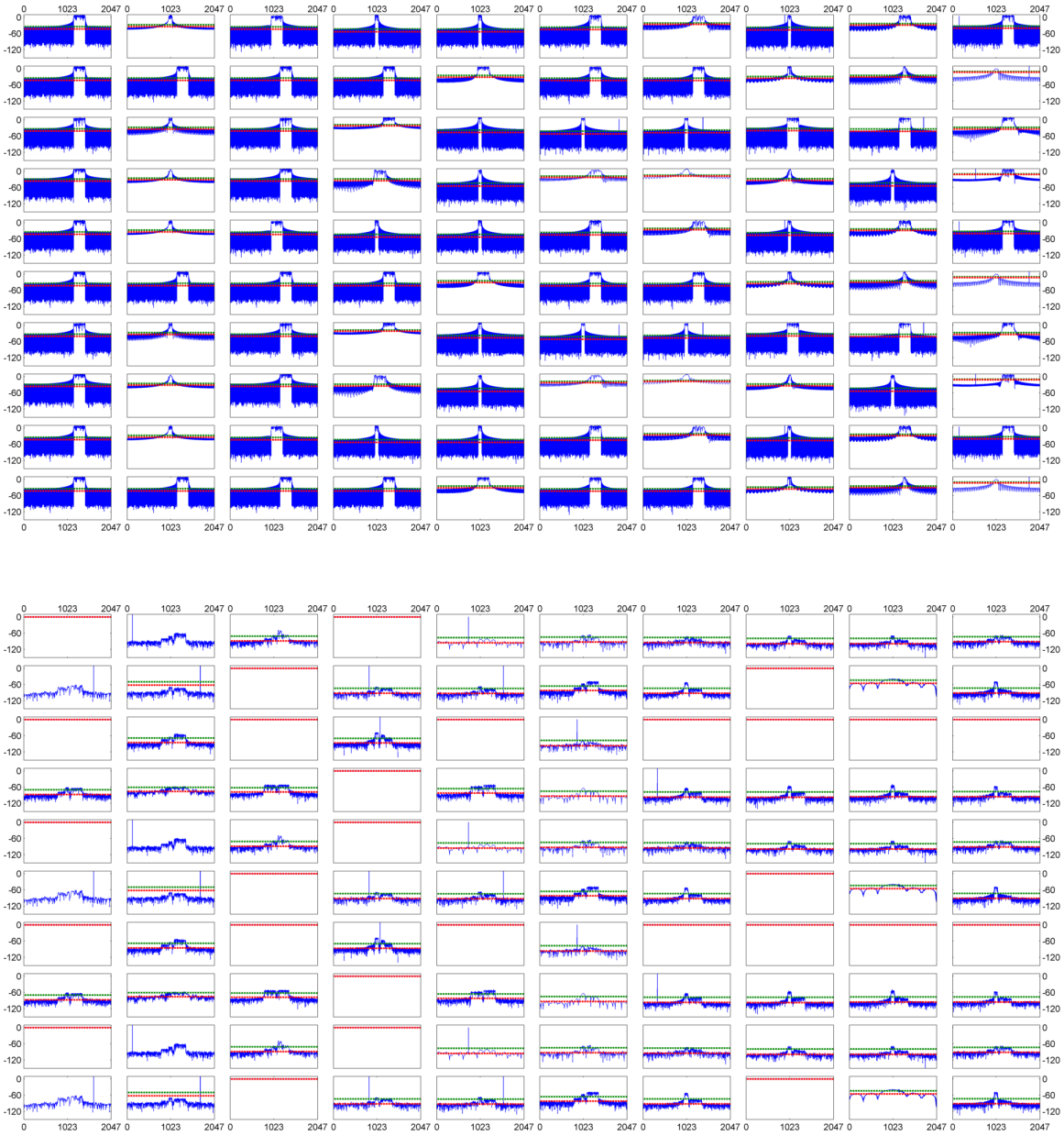


Figure 3.12: RF spectrum map examples.

- *Number of FFT bins.* This parameter affects determination of the REMs as the number of FFT bins is also the number of bits in a REM.

- *REM threshold coefficient (α).* Given that each map contains a unique energy density at each bin, the REM threshold coefficient sets the REM threshold (see Figure 3.6) α percent above the mean of the energy levels in that spectrum. The equation for computing the REM threshold using the threshold coefficient α is shown in 3.1, where N is the number of bins, and S represents the vector containing all N energy density values. For this research, spectral density values are recorded in decibels and are negative in sign. The value for α is limited to the range $[-0.5, 0.5]$.

$$Threshold = (1 - \alpha)\frac{1}{N}\sum_{i=1}^{N} S_i \tag{3.1}$$

This research uses a coefficient of 0.2 as in Figure 3.13, which is similar to the original threshold coefficient example in [35]. If, in a given bin of the FFT, the signal

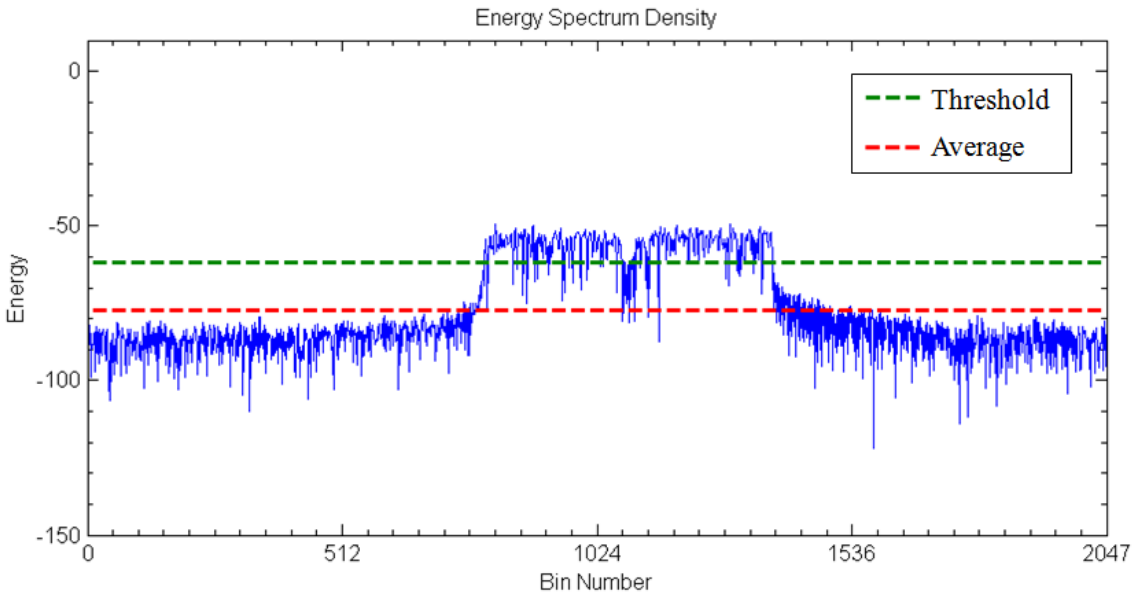


Figure 3.13: Applied threshold coefficient example.

exists above the threshold, the corresponding bit of the resulting REM is marked as a '1' vice a '0' otherwise. This technique is applied in Section 3.2.5.4 when computing the performance metric and in Section 3.3.4.2 for map merging.

- *k-means heuristic.* Exposing the different heuristics allows for a decision on which heuristic, if any, outperforms the others and should therefore be integrated with the rest of the middleware architecture.

- *Number of k-means centers.* It is expected the number of k-means centers affects how many nodes will belong to a cluster, thereby influencing the commonly-available spectrum throughout.

- *Node distribution standard deviation (σ).* The clustering software accepts a number of parameters, including the standard deviation used in generating several distributions (including Gauss). In general, increasing this value increases the area covered by the Gauss distribution.

Table 3.2: System parameters.

System Parameter	Description
RF spectrum maps	Represents the maps used to simulate varying spectrum environments. Figures A.1 and A.2 shows two examples of ten-by-ten spectrum measurement grids where the top left plot corresponds to position (1,1) and the bottom right plot to (10,10). Thresholds are denoted with a red dashed line.
Number of FFT bins	The number of bins in an FFT of the spectrum. The value used for this experiment is 2,048 due to the equipment configuration.
REM threshold coefficient	Determines how much the mean energy level is adjusted when evaluating the spectrum for availability.
k-means heuristic	The different heuristics by which k-means clustering is performed. These heuristics are listed and briefly described in Section 2.4.3.
Number of k-means centers	Defines how many centers should be used by the specified k-means heuristic.
Node distribution σ	Provides a non-default standard deviation for determining several distributions. Pilot experiments show that $\sigma = 0.3$ yields Gauss distributions that consume roughly the same area as the other distribution types.

3.2.7 Factors.

Arguably the single most important parameter is the RF spectrum map under which a node map exists. If the spectrum map is held constant, it is useless to assert clustering produces expected ICSS results. Thirteen total RF spectrum maps are used to validate clustering algorithm performance. Additionally, four clustering heuristics and seven center counts are used.

Table 3.3: System factors.

Factor	**Levels**
RF spectrum maps	(see Figures 3.14 and 3.15)
Number of maps	10
Node distributions	Uniform, Gaussian ($\mu = 0$), Multi-Cluster ($k = 2$).
k-means heuristic	Lloyd's, Swap, EZ-Hybrid, Hybrid
Number of centers	2 to 98, in intervals of 2 (49 total)

Ten RF spectrum measurement grids are used to validate the system's performance across a range of transmitter configurations. These spectra are emulated and their FFTs taken ("snapshots" of the spectrum) at every whole-number coordinate within the specified dimensions (i.e., $(4, 5)$). The REMs formed from these snapshots are used in determining ICSS. All ten spectra are shown in Appendix A.

For the non-baseline data, the spectrum at each grid location is generated via spectrum emulation equipment where a custom transmitter map may be specified. The transmitter maps used to generate the RF spectra for this experiment are shown in Figures 3.14 and 3.15. In these diagrams, the map has dimensions 10×10 such that each has a maximal coordinate of $(10, 10)$. Triangles represent transmitters, large circles represent high-power

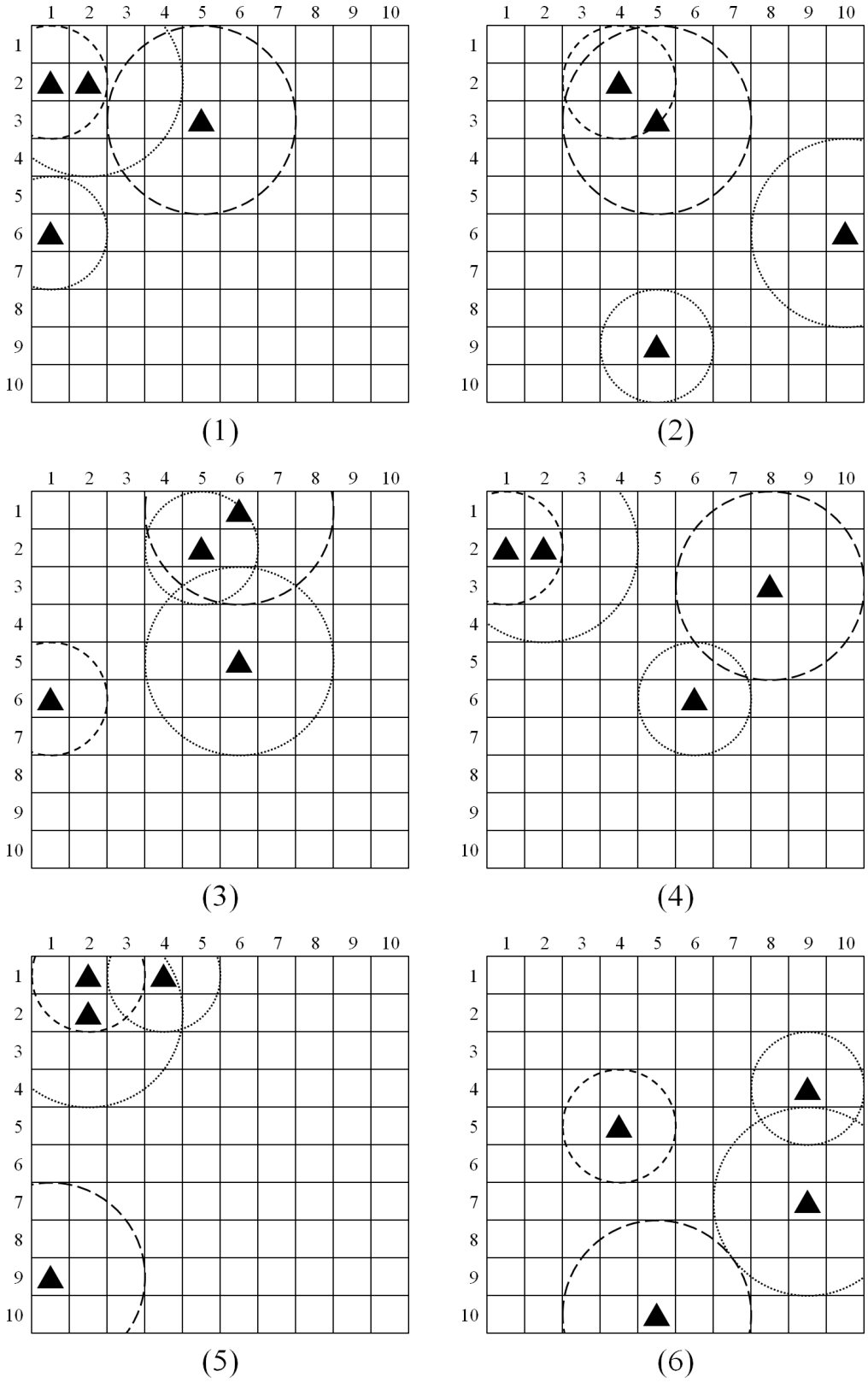


Figure 3.14: Transmitter configurations 1–6.

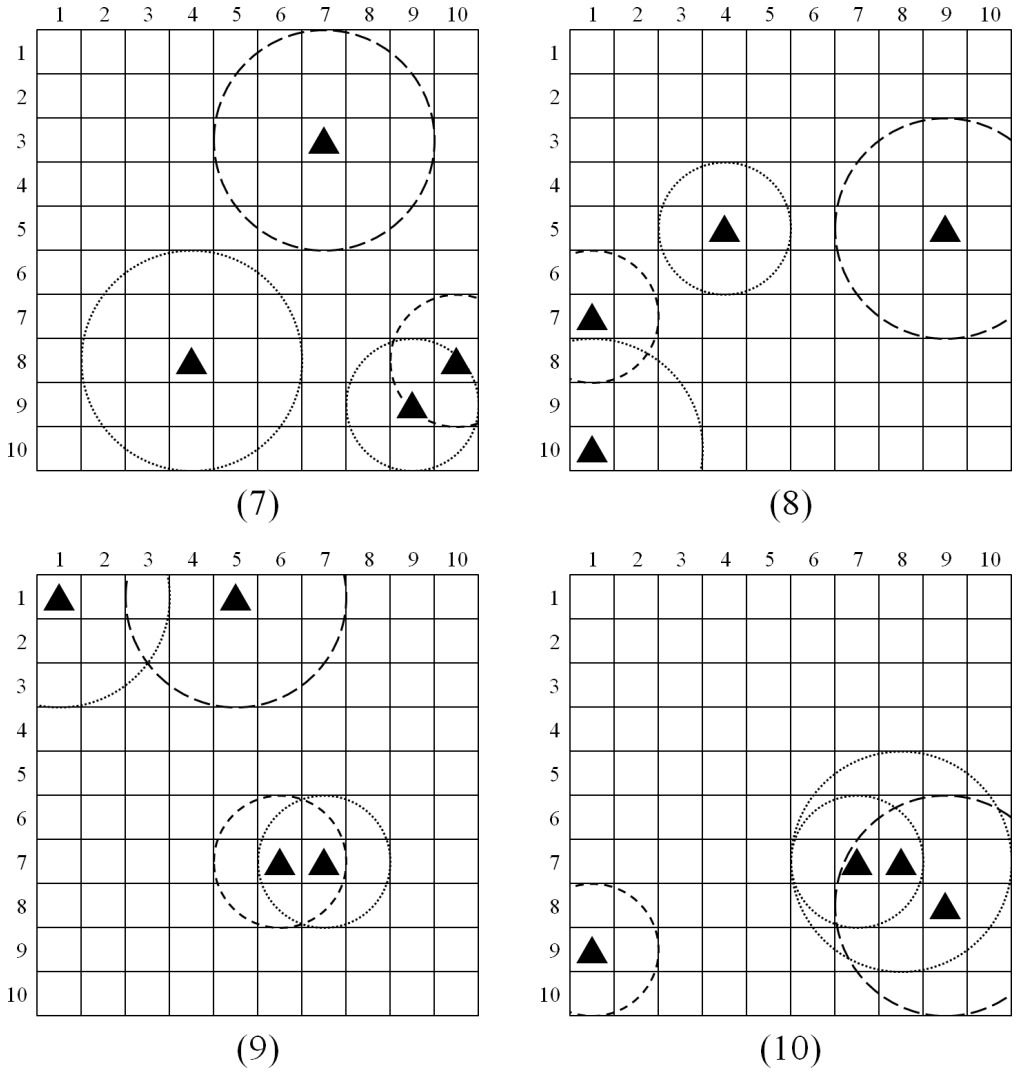


Figure 3.15: Transmitter configurations 7–10.

transmitters (as opposed small circles for low-power), and dashed lines represent wide-band transmitters (as opposed to dotted lines for narrow-band).

We use `kmltest.exe` to generate ten different receiver maps of three distribution types. Each map is pseudo-randomly generated using integer seed. Because the program is known to crash when using seeds of three and seven, the seed number is incremented twice upon reaching those numbers such that they are not used. One map of each distribution

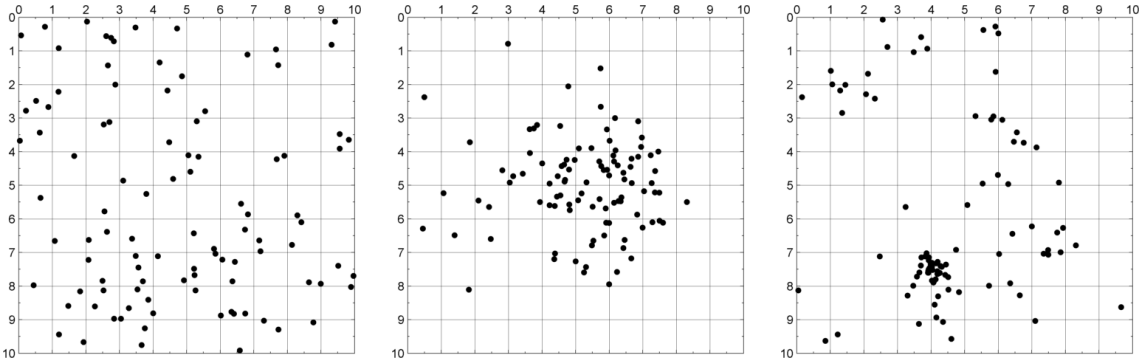


Figure 3.16: Sample uniform (left), Gauss (center), and multi-cluster (right) node distributions with seed = 1.

type (seed = 1) is shown in Figure 3.16 where the *x*- and *y*-axes correspond to the nodes' respective physical locations. All ten maps (using seeds {1, 2, 4, 5, 6, 8, 9, 10, 11, 12}) are shown in Appendix B.

The *k*-means testbed described in Section 2.4.3 features four clustering heuristics. Therefore, all four are used for this experiment for exploring any variability and/or improvement among heuristics prior to incorporation with the system.

All center counts are powers of two for binary division of the map. We use this rule to facilitate simpler creation of a more vertical hierarchy in future work. Additionally, pilot experiments revealed that `kmltest.exe` crashed when the number of centers was equal to the number of nodes. Therefore, the maximum number of centers is the highest power of two less than the number of nodes; in this case, 100.

To emulate the ability of nodes to organically sense the RF spectrum, nodes are mapped to DYSE spectrum measurements. When nodes are assigned spectra for clustering heuristic evaluation, they receive the spectra assigned to the grid cell in which they are located, per Figure 3.16 and the rest of the maps located in Appendix B. The grid cell is the whole number of the node's coordinates. For example, a node with a location

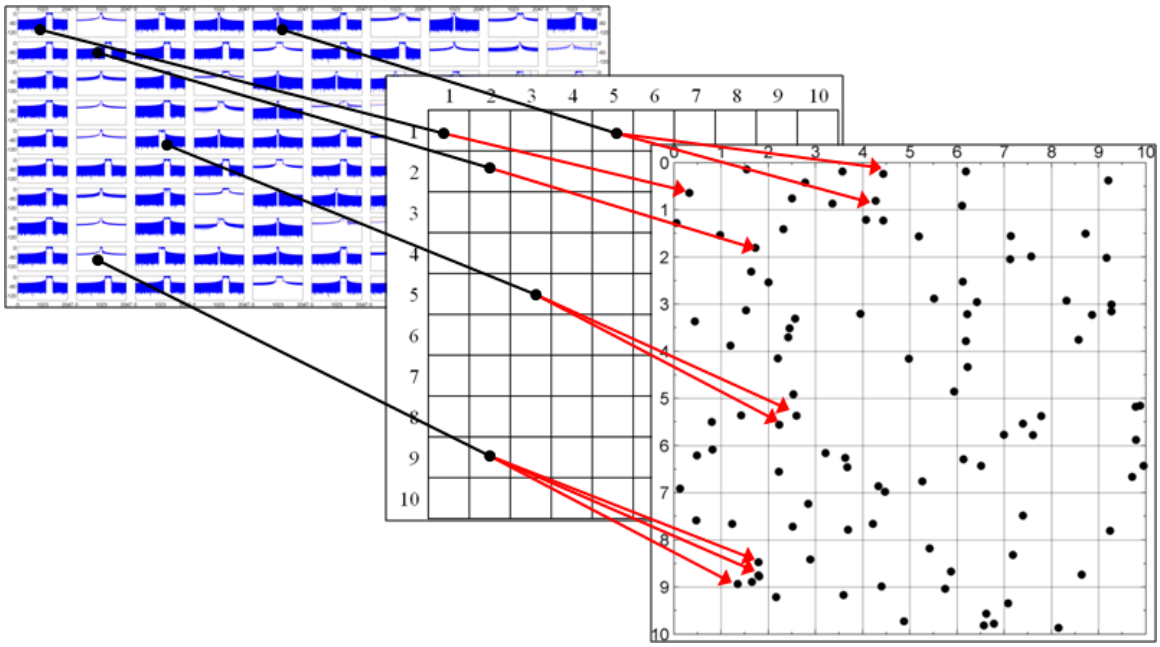


Figure 3.17: Spectrum-to-node mapping example.

(4.23288, 7.31245) will be assigned the spectra of grid cell (4, 7). A demonstration of this mapping of spectrum measurements to nodes is shown in Figure 3.17. Black lines show how spectrum measurements map to individual grid cells, and red arrows show how those grid cells with spectrum measurements are mapped to the nodes within each grid cell.

3.2.8 Evaluation Technique.

3.2.8.1 Technique.

Evaluation is via simulation followed by analysis. It is not practical to evaluate this system purely by simulation because the ICSS metric must be analyzed using simulation output.

3.2.8.2 Experimental configuration.

Host machine. This work is performed on a laptop containing an Intel Core i7-2720QM containing eight cores running at 2.20 GHz. The machine has 16.0 GB of RAM and runs Windows 7 with Service Pack 2.

Software and code. MATLAB is used to build this test framework, run the simulation, and analyze output. All associated code is included in Appendix F.

3.2.8.3 Results validation.

Results are validated by examining (a) whether clustering does indeed behave as expected using the baseline maps, (b) whether the ICSS value does asymptotically approach unity as the number of clusters approaches the number of nodes, and (c) whether the ICSS value is consistent between distribution types and across different node distributions and RF maps. If the two latter observations hold throughout the experimental results, the *k*-means algorithm is a solid fit for use in a network where the network's configuration depends on the RF spectrum environment. Additionally, if one heuristic consistently performs better than Lloyd's algorithm, then that heuristic is deemed the primary choice for use in such a network. If no heuristic does so consistently or at all, then Lloyd's algorithm is chosen for implementation. This validation approach is visual confirmation based on a large dataset, so it is expected any trend is distinct and readily identifiable by the human eye.

3.2.9 Experimental Design.

This experiment is full factorial as the number of possible input combinations is feasible to implement. There are 58,800 total simulations (10 RF spectrum maps $\times$ 10 node distributions $\times$ 3 distribution types $\times$ 4 clustering heuristics $\times$ 49 center counts).

3.2.10 Methodology Summary.

This experiment is designed to evaluate the effectiveness of several different clustering algorithms for incorporation into a cognitive radio system. Experimental data and analysis is generated using MATLAB code, and results are compared using the proposed new metric, ICSS.

3.3 Adaptive Hopset Selection

3.3.1 Spectrum Input.

Spectrum sensing is the first step in forming the adaptive hopset. First, a radio front-end receiver (Rx) senses the spectrum. An analog-to-digital (ADC) converter converts this signal to its digital equivalent. Next, a Fast Fourier transform (FFT) is performed on the digitized signal to generate an N-bin frequency-domain model of the signal. The FFT output is then converted to a binary vector corresponding to the spectrum representation definition presented in Section 3.2.5.1. The entire spectrum input process is shown in Figure 3.18, where components designed as part of this section are shown in red.

The portion of Figure 3.18 in dashed lines shows the REM distribution process and the centralized REM-based network clustering process. Once clusters exist, common REMs can be formed for individual clusters and the network as a whole using the same merging process. Clustering is not a focus of this paper, however, and will not be explained further here.

In this section, we assume spectrum sensing is complete for the network, that every node can represent its REM in this manner, and that all nodes have received the REMs of

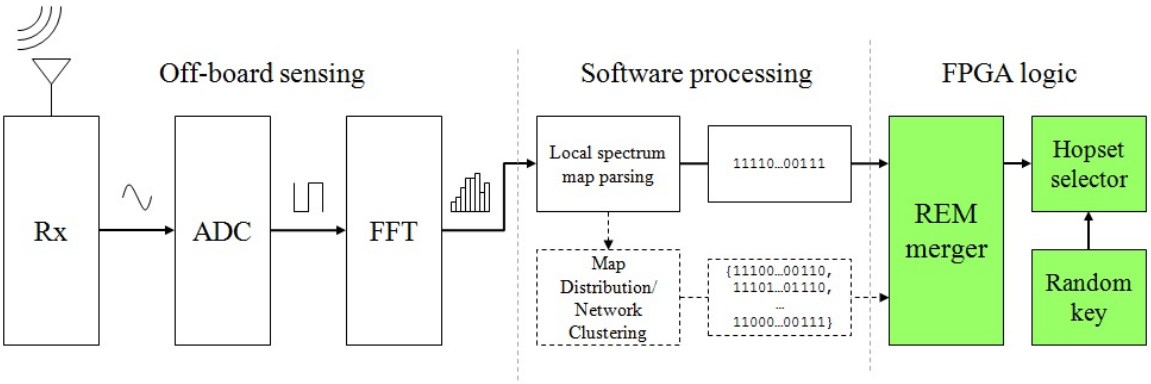


Figure 3.18: Spectrum input diagram.

all nodes in the network. In terms of Figure 3.18, we assume that all steps prior to the REM Merger block are complete and REM vectors are ready for merging.

3.3.2 FPGA Internal Structure.

We implement our design on a Xilinx Virtex IV FPGA as part of the WARP II radio transceiver board. Using the Xilinx Embedded Development Kit (EDK), we assemble a collection of custom and existing intellectual property (IP) cores using a 32-bit bus and the FPGA's embedded PowerPC processor. In Figure 4, green cores are affiliated with the PowerPC, blue cores are WARP-specific, tan cores are those needed to support our IP core, and our core is shown in red. Cores with an asterisk (*) are not used in this experiment, but will be used in future development.

The IP cores supporting our core are RS-232 (for output to PC serial port terminal) and a removable CompactFlash memory. The latter two IP cores are grouped together by a gray dashed line because they are interchangeable as method for exchanging REMs. Initially, we simply write each node's REM to the removable memory and transfer REMs between nodes in this fashion.

Future iterations of this design will include transmit REM data via wired network for system testing purposes. Finally, radios will operate in a completely wireless fashion using their radio cards. The bus structure of our design is shown in Figure 3.19.

3.3.3 IP Core Internal Structure.

Our IP core consists of six components: REM merger, key loader, two 64x32 register files for storing the aggregate REM and the key, the hopset selector, and a BRAM section in which open channels are stored. Their organization is shown in Figure 3.20. Buses are shown in bold, and bus widths are in square brackets. A larger version of 3.20 is shown in Appendix C as Figure C.2. All VHDL code necessary to implement this IP core is included in Appendix G.

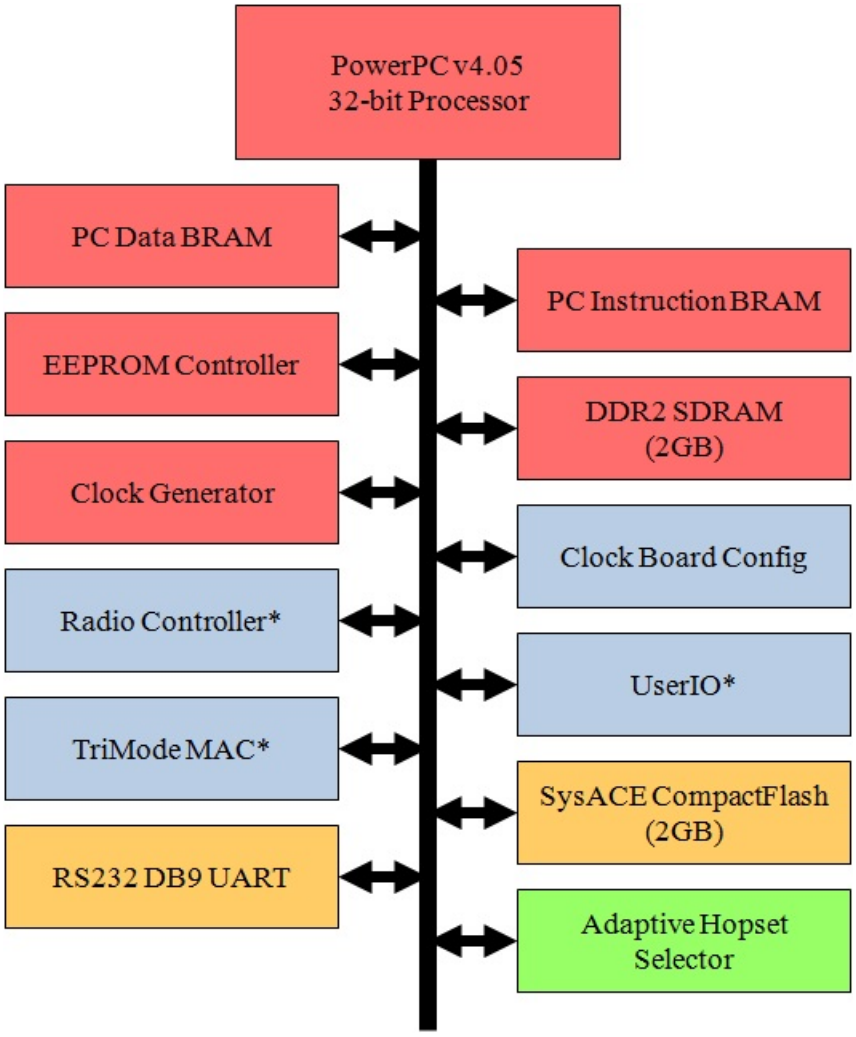


Figure 3.19: FPGA bus structure diagram.

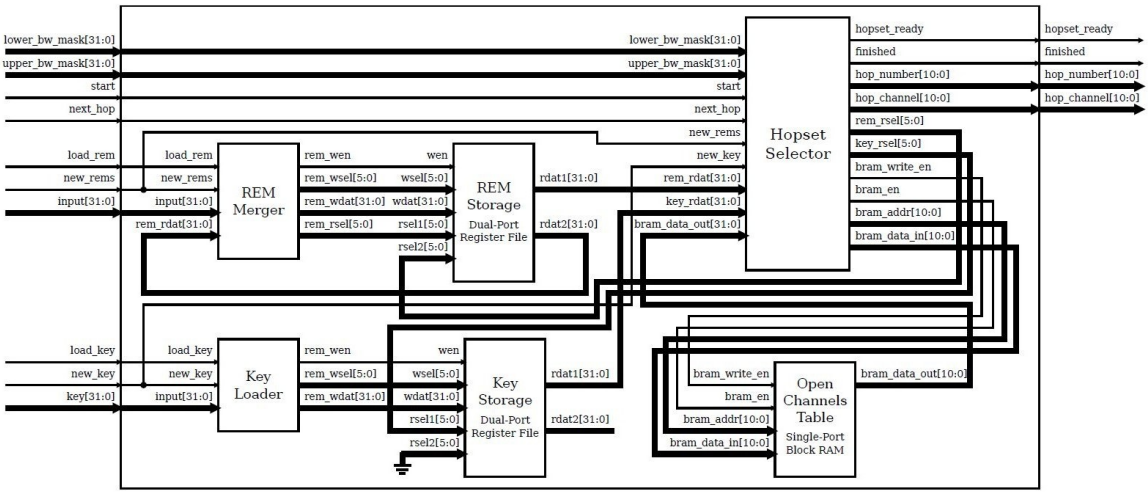


Figure 3.20: AHS structural diagram.

3.3.4 IP Core Internal Function.

3.3.4.1 Bandwidth Masking.

Different radio protocols have different bandwidth requirements. In order to provide flexibility with regard to these requirements, our device accepts two 32-bit bandwidth masks. When the hardware scans the aggregate REM for available channels, the two masks are applied to determine whether the required band is available. For example, if a given protocol requires B bins to be available around a center frequency, the user specifies the corresponding bits in the bandwidth mask input vectors as 'high'. The construction of the bandwidth mask is shown in Figure 3.21.

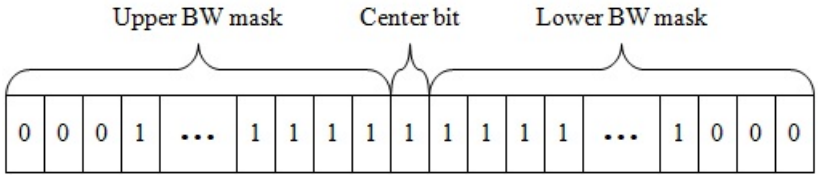


Figure 3.21: Example bandwidth mask vector.

3.3.4.2 REM Merging Components.

REM merging is the first core function of this design. In this paper, we assume that each REM is a 2048-bit binary vector, the direct result of a 2048-bin FFT. REMs can be composed of a variety of types of spectrum data; we use only spectral data represented as binary vectors per Section 3.2.5.1.

Further, REM merging is the process of successive bitwise AND operations. In this manner, the AND of any number of REMs produces a new REM for which available channels are the intersection of all available channels across all input REMs. Each vector is loaded successively and ANDed with the previous result per the merging technique presented in Figure 3.22. There is no physical limit to the number of input REMs, although the number of possible REMs is ultimately limited to the network size. All vectors are serially loaded at a rate of one word per clock cycle. The number of clock cycles needed to load a map is modeled by Equation 3.2.

$$Cycles = \frac{\#REMs}{map} \times \frac{Bbits}{REM} \times \frac{1cycle}{Buswidth} \tag{3.2}$$

For example, consider the scenario in which 64 REMs are used. A 64-bit map with 32-bit bus will take $\frac{64REMs}{map} \times \frac{2048bits}{REM} \times \frac{1cycle}{32bits} = 4,096cycles$ to load all maps. If the number of

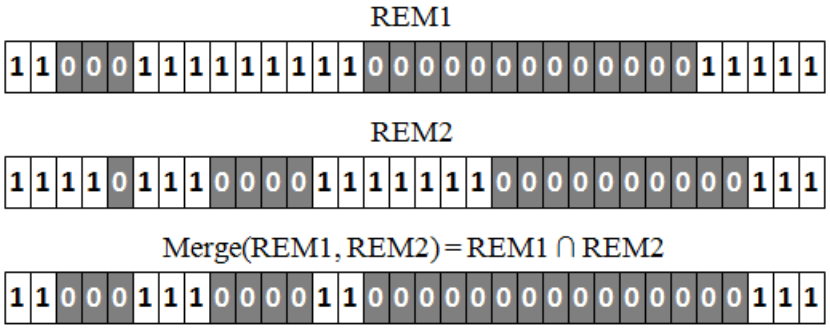


Figure 3.22: Map merging example.

vectors is less than the total number of possible input gates, unused input gates are assigned all ones such that the bitwise AND of any vector with that of all ones results in the original vector. All vectors are serially loaded at a rate of one word per clock cycle.

The REM merger can accept an arbitrary number of maps as the hardware component logically ANDs each 32-bit input with its respective section in the aggregate REM. For example, if the nth 32-bit section of a REM is loaded into the REM merger, it will be ANDed with the nth section of the REM stored by the merger. The result is then written to the register storing that section of the aggregate REM. In this way, any number of devices in a network can share REMs for the purpose of creating an adaptive hopset.

3.3.4.3 Adaptive Hopset Selection.

Adaptive hopset selection, or AHS, represents the second core function of our design. Map merging generates the aggregate REM as a binary vector in which each bit maps to the center frequency of an FFT bin within the device's usable spectrum. Because all radios within the network know this mapping, adaptive hopset selection becomes a matter of randomizing these channels. We assume the hopset S is H hops in length, bit position-to-channel mappings are stored in the device (i.e., in a look-up table), and an N-bit random key K exists prior to forming the hopset. Under these assumptions, our circuit performs the following steps on an N-bit REM to convert the map into a corresponding adaptive hopset:

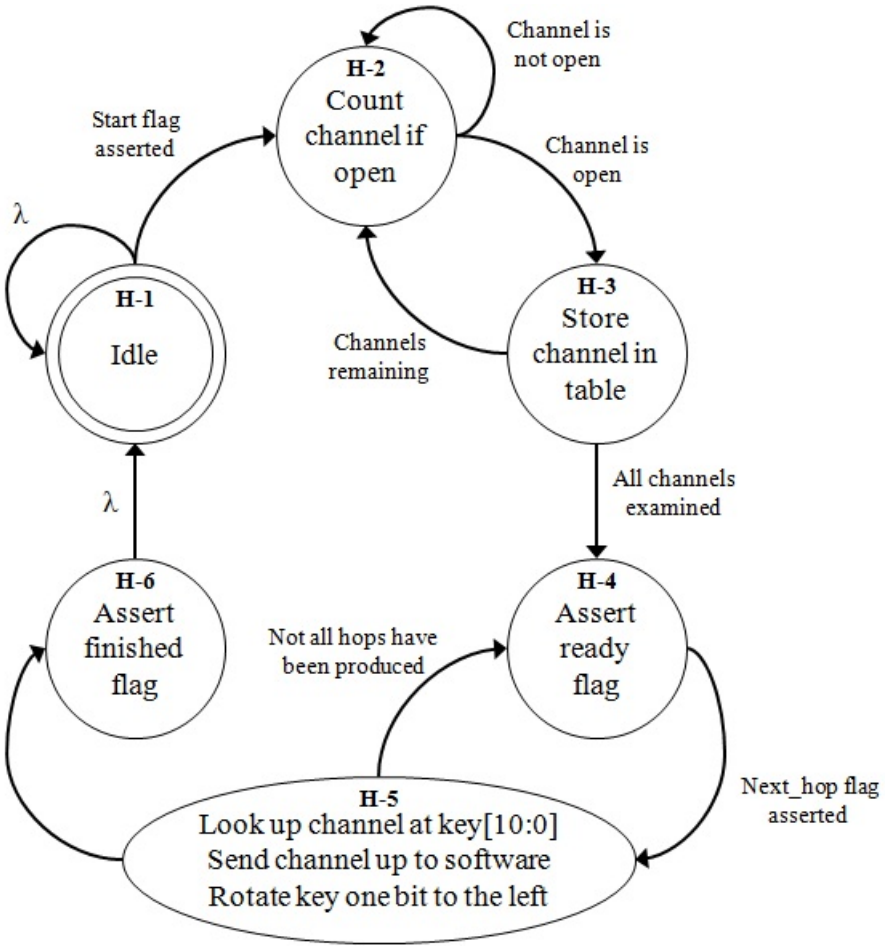


Figure 3.23: AHS functional diagram.

1. Starting at the lowest-number channel, begin iterating through the binary vector. Initialize the available channel count $C = 0$.

2. For every bit, if the bit is a '1', look up the corresponding channel number and store it in a list L containing available channels and increment C.

3. If $C < H$, replicate the first $H - C$ at the end of the list L such that L has H elements.

4. For $i = 0$ to $i = H - 1$, look up the available channel in L at the index indicated by the key K, scaled to the number of open channels. Output the channel in L as the i^{th} hop in hopset S. Rotate K one bit to the right.

5. Repeat step four H times, i.e., until all H hops have been assigned within S.

The hopset S of length N is this system's output. Because each index is derived from a random number, every index is inherently random. This process is graphically illustrated as a finite state machine in Figure 3.23.

3.3.5 System Testing.

VHDL code was developed in ModelSim PE. Once we developed a consistent simulation design, we synthesized the hopset selector core in the Xilinx ISE implementation tool. We used the mapped version of the circuit to confirm the design's functionality with the effects of realistic delays and other hardware characteristics prior to loading the integrating the circuit with the rest of the WARP system. We show the results of this simulation in the Results section.

We tested the hardware system using C code implemented on the FPGA's embedded processor core by reading in the spectrum maps from a compact flash file, submitting the maps to the VHDL code described above, and writing the resulting hopset(s) to a new compact flash file to simulate REM reception and hopset transmission over the network. We use random maps and a random key to ensure a random hopset is indeed generated from a realistic input.

3.3.6 Optimization Goals.

Virtex IV FPGA resources are already limited, and incorporating the existing WARP system causes the design to be even more crowded. Therefore, this design is optimized for area. Because the state machines and other logic in our IP core require relatively little area (on the order of 5% total resources), our primary optimization was to minimize the effort needed to synthesize and implement memory. This strategy implicitly attempts to use memory components built into the board. We used a generic VHDL block RAM (BRAM) structure given in the XST User Guide for storing available channels [41].

4 Results

4.1 Network Clustering

4.1.1 Clustering Baseline.

In order to verify that clustering does behave as expected, four canned node distributions were generated such that clusters were readily apparent. The *k*-means algorithm was run on example distributions containing two, four, eight, and 16 clusters. Sixteen clusters was the upper limit as it was assumed the trend would continue in the same fashion as the first four greater-than-two binary counts. As shown in Figure 4.1, each heuristic clustered the distributions as expected (i.e., two clusters were formed in the map

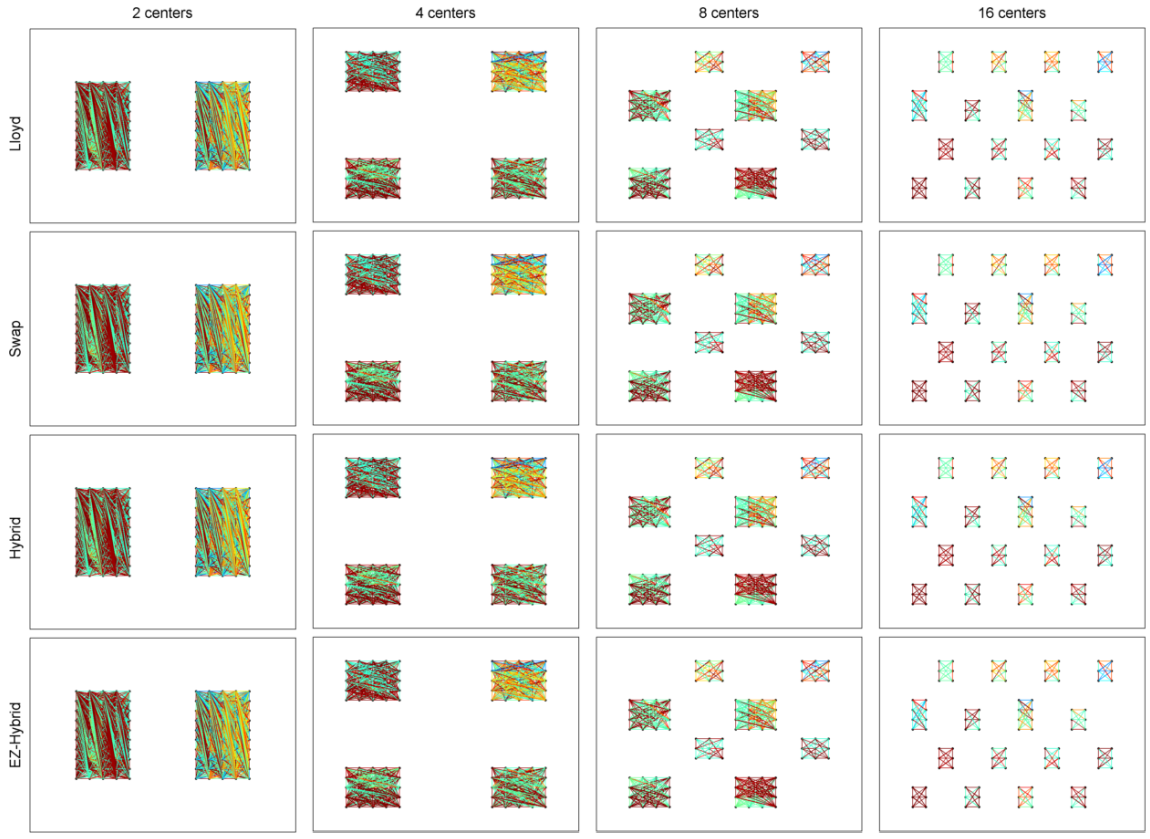


Figure 4.1: Canned cluster verification test.

containing two distinct clusters, etc.). Different line colors are irrelevant to this portion of the experiment.

This proof-of-concept validates the simple (but very important) hypothesis that the clustering testbed software does perform as expected. As a result, it is assumed all results obtained using the clustering software are accurate representations such that any obscurities in the results pertain solely to the spectrum-based clustering metric.

4.1.2 Clustering Visualization.

Lines are drawn between all nodes within each cluster to represent potential (pre-routing) communication links. When nodes within a cluster have highly similar REMs (i.e., a high ICSS value), the connecting links are colored red. Likewise, links between nodes with dissimilar REMs are colored blue. This is accomplished by selecting colors from the MATLAB `colormap()` function according to the inter-node ICSS value. The clustering visualization method show only center counts of two, four, eight, 16, 32, and 64 in the interest of preserving space. Figures 4.2, 4.2, and 4.2 show examples of this cluster visualization method using the third DYSE map on all three distribution types.

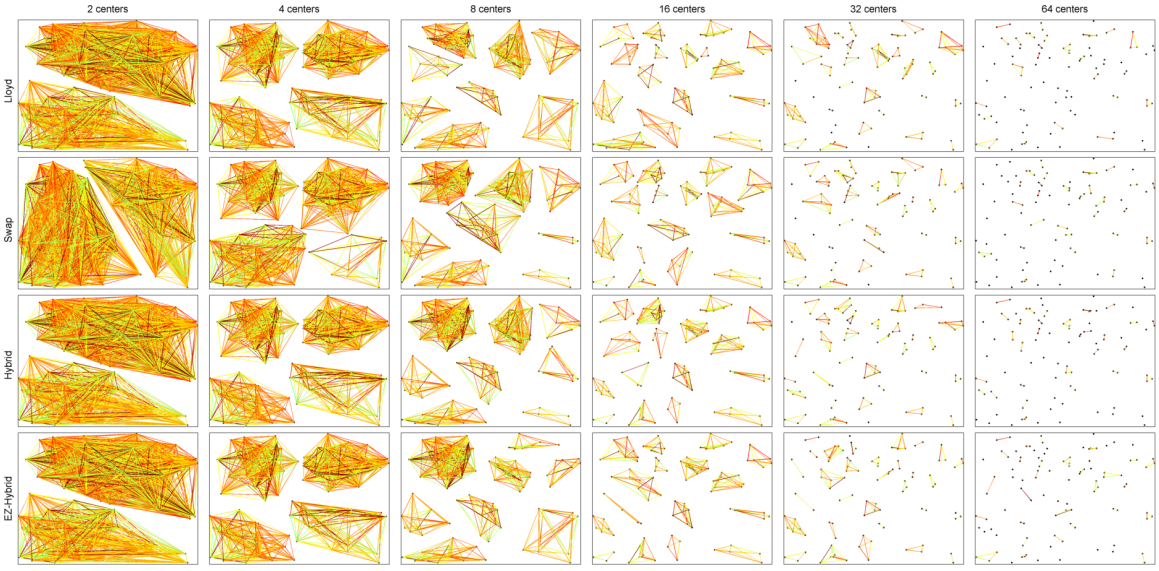


Figure 4.2: Cluster visualization of a uniform distribution using DYSE map #3.

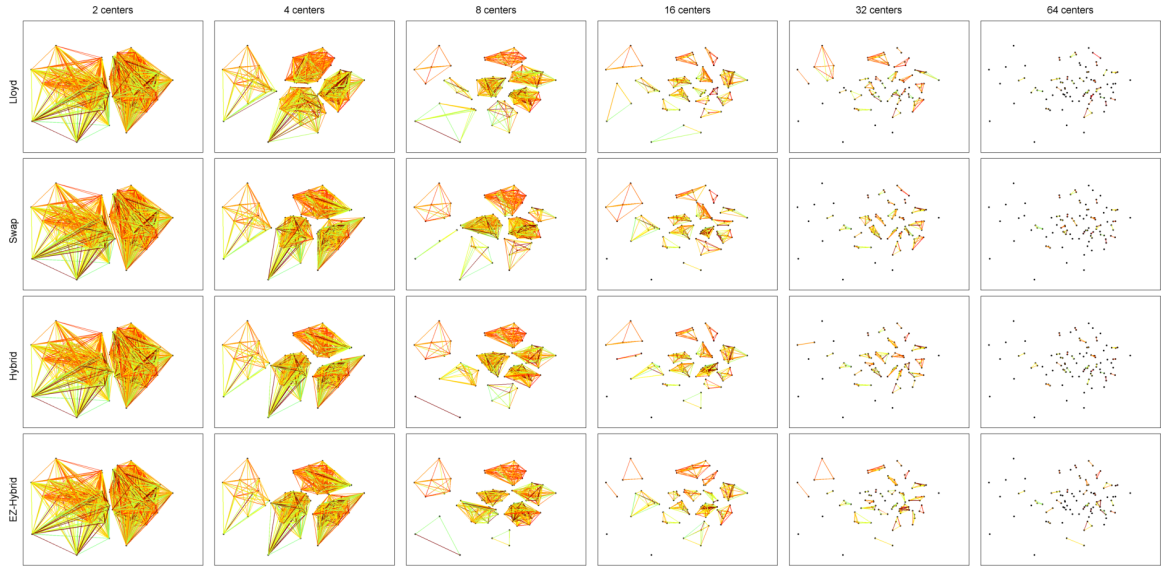


Figure 4.3: Cluster visualization of a Gauss distribution using DYSE map #3.

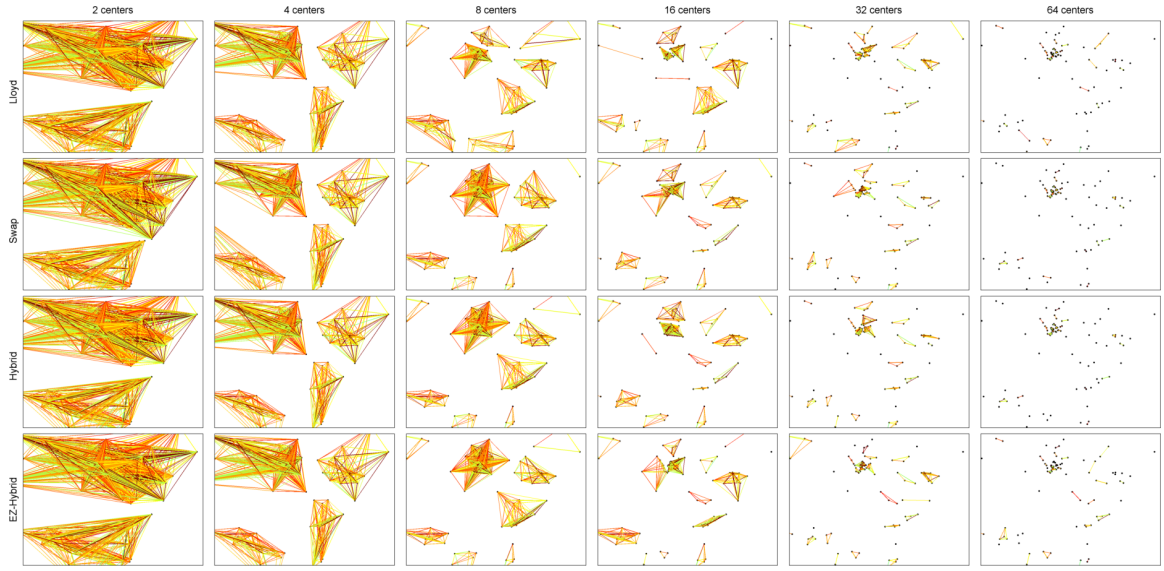


Figure 4.4: Cluster visualization of a multi-cluster distribution using DYSE map #3.

Multiple cluster visualizations are presented in Appendix D. Each distribution type is used ten times, and each pseudo-random distribution (based on a random seed) is used three times. This configuration yields a total of thirty different cluster visualizations.

4.1.3 ICSS Evaluation.

ICSS plots for the first and fifth DYSE maps are shown for all three distributions in Figures 4.5, 4.6, and 4.7. ICSS is found to vary between approximately 60% and 100% across the range of clustering heuristics and distribution types. Each datapoint represents running the relevant clustering heuristic on the ten distributions shown in Appendix B and averaging the ICSS result. It should be noted that while the trends do look nearly identical, close inspection reveals a slight yet distinct difference between plots.

Five characteristics are readily apparent in all three ICSS plots. An asymptotic relationship to unity at high cluster counts, a dip at lower counts in uniform and Gauss distributions, the inconsistency of the EZ-hybrid heuristic at cluster counts over 50 in a 100-node distribution, the relative consistency of both the Lloyd and Hybrid heuristics over (nearly) the entire range, and the immediate drop-off in ICSS for the Swap heuristic at high cluster counts are all plainly visible on first inspection. These characteristics are examined below.

The asymptotic relationship over the range of cluster values is as expected and confirms the hypothesis that spectrum similarity between nodes within the same cluster asymptotically approaches unity as the mapping of nodes to clusters becomes one-to-one. In addition to confirming this trend, the radio can also be pre-loaded with a range of clustering results such that different cluster counts are available for selection based on desired ICSS. For example, if the RF environment contains a significant amount of interference, nodes likely need a higher ICSS value in order to increase the likelihood that a common hopset can be found. Per the results of evaluating the ICSS metric, this need translates to more clusters. Conversely, an RF environment with little interference requires less commonality (i.e., lower ICSS) for approximately the same number of common frequencies.

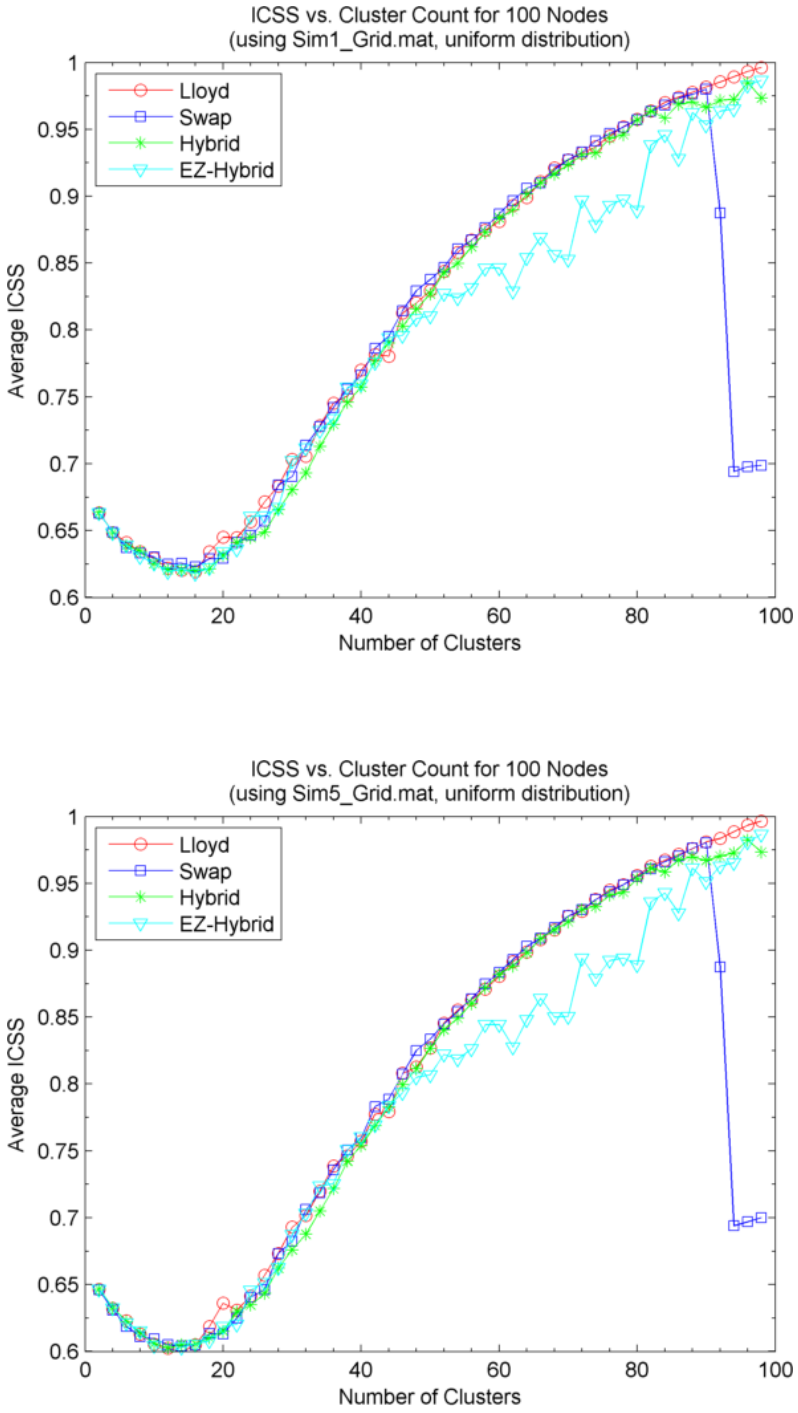


Figure 4.5: ICSS for uniform distributions using DYSE maps #1 and #5.

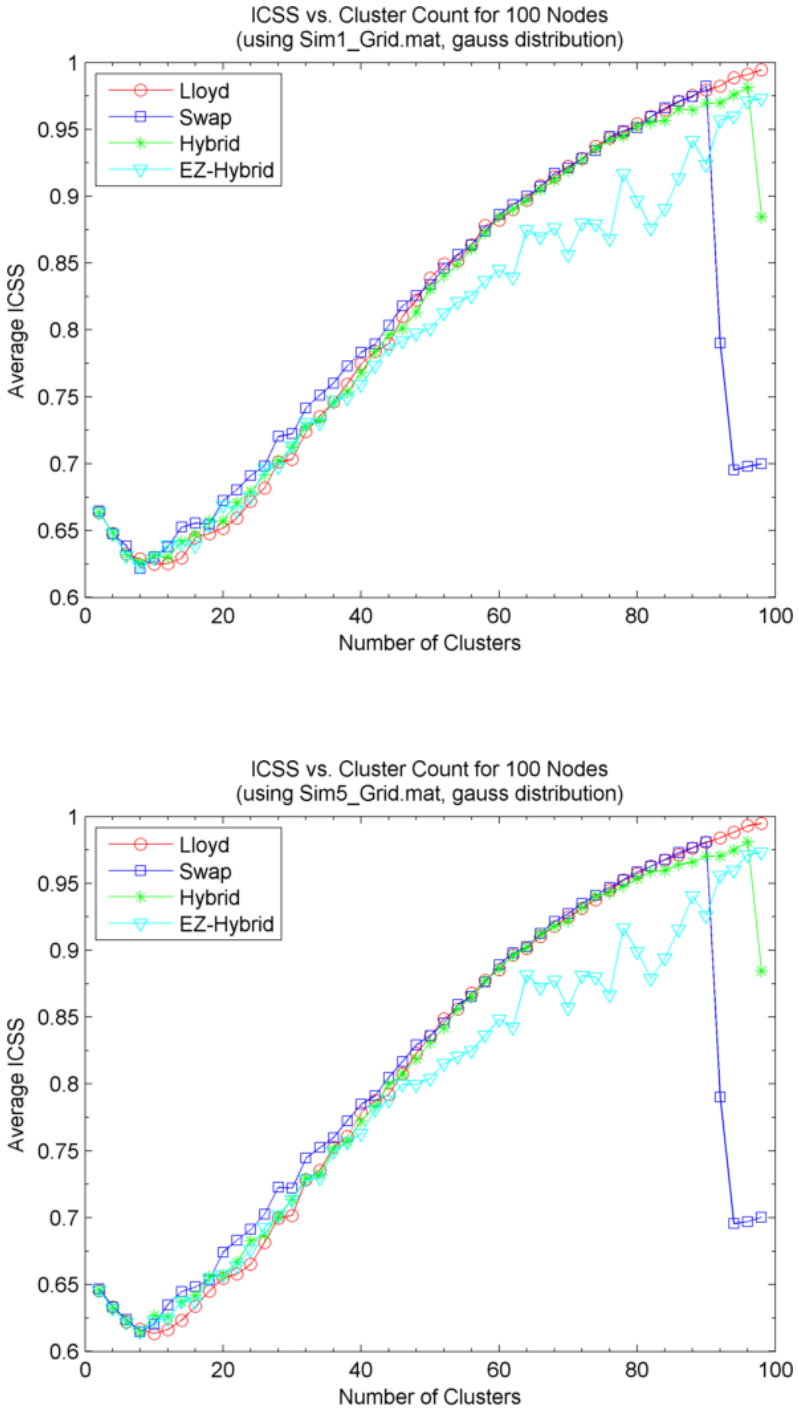


Figure 4.6: ICSS for Gauss distributions using DYSE maps #1 and #5.

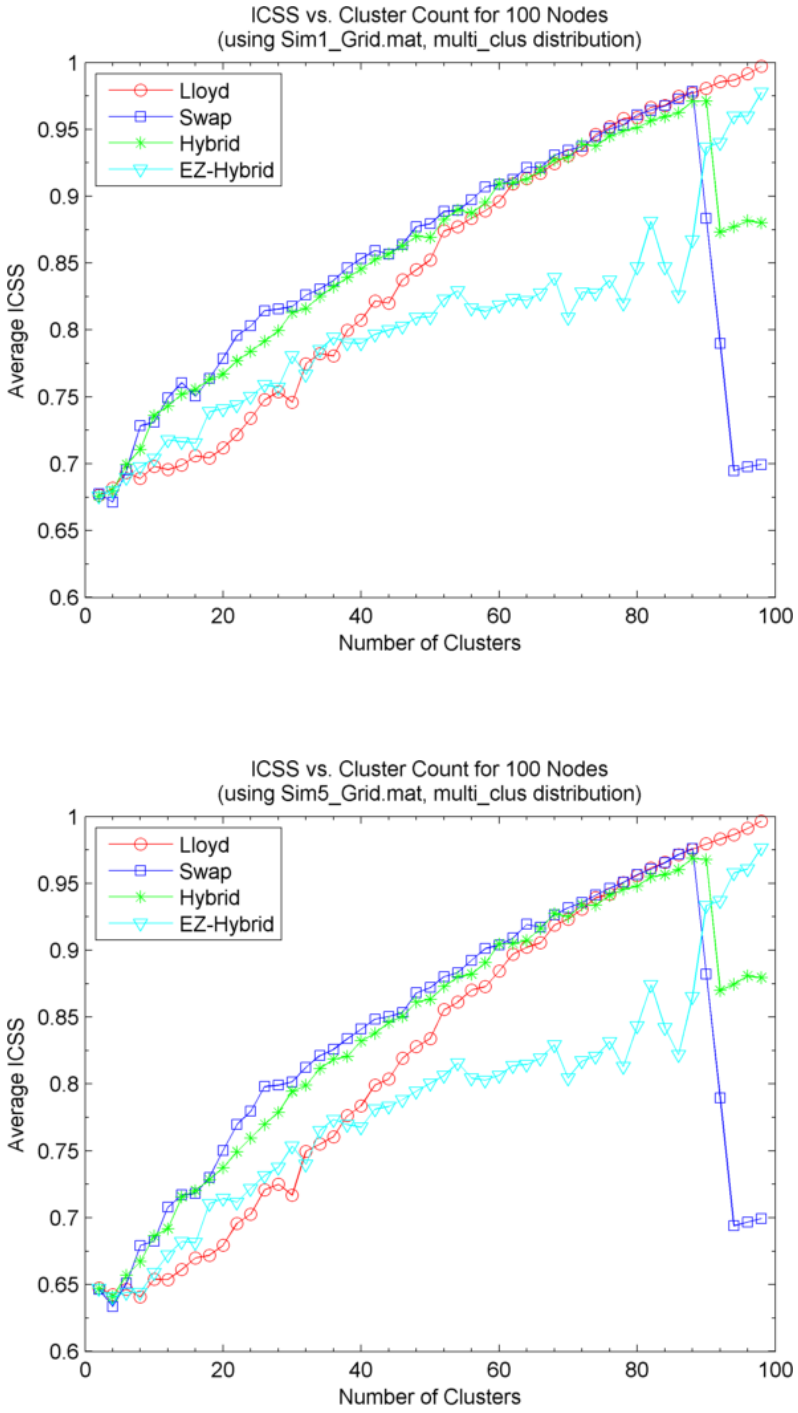


Figure 4.7: ICSS for uniform distributions using DYSE maps #1 and #5.

There exists a dip in ICSS occurs around 16 clusters, with the ICSS value remaining below that of two clusters until the number of clusters reaches approximately 25. More testing is needed to explore whether ICSS "breaks even" at 25 (in this case, 25% of the total number of nodes), and why this occurs in the uniform and Gauss distributions, but not in the multi-cluster distribution.

The EZ-hybrid heuristic experiences a series of oscillations at higher cluster counts (i.e., those counts greater than half the total number of nodes). This is contrary to the monotonic trend in the three other heuristics (aside from Swap at high counts). This is because EZ-hybrid is a simple hybrid of the Lloyd and Swap heuristics and does not make any effort to optimize clustering. The Hybrid heuristic, on the other hand, does not show any such dropoff except at the highest cluster count. This result, however, is because the test program crashes with regularity at that cluster count.

Similarly, the Swap heuristic crashes for each of the last four cluster counts. Because of this instability, the Swap heuristic appears to be a poor choice for implementation with the remainder of the system. If it not for the consistent crash tendency at high cluster counts, Swap performs evenly with (and often better than) the three other heuristics. However, if the desired ICSS is lower than approximately 97% (a very likely scenario), the Swap heuristic is an ideal choice.

4.1.4 Application to System Implementation.

This experiment highlights the tradeoffs present in selecting the number of clusters chosen for a CRN implementation. From Figures 4.5, 4.6, 4.7, and those in Appendix E, it is apparent that increasing the number of clusters yields higher bandwidth. Further, partitioning the nodes into many clusters decreases network complexity and limits excess traffic loads on the overall network as intra-cluster communication is independent of other clusters. However, while high bandwidth potentially means less interference, it is not always necessary. Networks experiencing few or no occupied channels require

less commonality between nodes, which leads to a lower ICSS value and fewer clusters. Therefore, these results can be used to retrieve in real-time a desirable cluster count given a minimum or maximum ICSS value.

The cluster count used as input to the clustering algorithm in actual operation is driven solely by the desired ICSS value, but ICSS is affected by the radio environment itself. We intend to incorporate the results of this research as a cache of experimentally-obtained data. A cache reduces computing time, and it is expected that the data obtained using 100 nodes can be extrapolated to both smaller and larger networks with the same result. Ultimately, using a table-lookup scheme balances the tradeoffs anticipated in different RF environments where greater interference can be offset by increasing the number of clusters in order to improve the similarity of spectrum observations between nodes.

4.2 Adaptive Hopset Selection

4.2.1 Mapped Simulation.

In order to validate our design before running the time-intensive place-and-route process and to facilitate quicker debugging, we simulated the synthesized/mapped version of our circuit. Figures 4.8-4.13 detail this simulation and reinforce our assertion that the circuit performs as intended given realistic hardware constraints. Signals of interest are described in captions. The synthesized component was instantiated in the same VHDL test bench used to evaluate our code base. Xilinx automatically generated the VHDL file using Unisim components which accurately model hardware constraints in simulation.

Initially, the `new_rems` and `new_key` flags are toggled to clear the existing aggregate REM and key. The REMs are loaded by writing the 32 bits of REM data and toggling the `load_rem` flag (Figure 4.8).

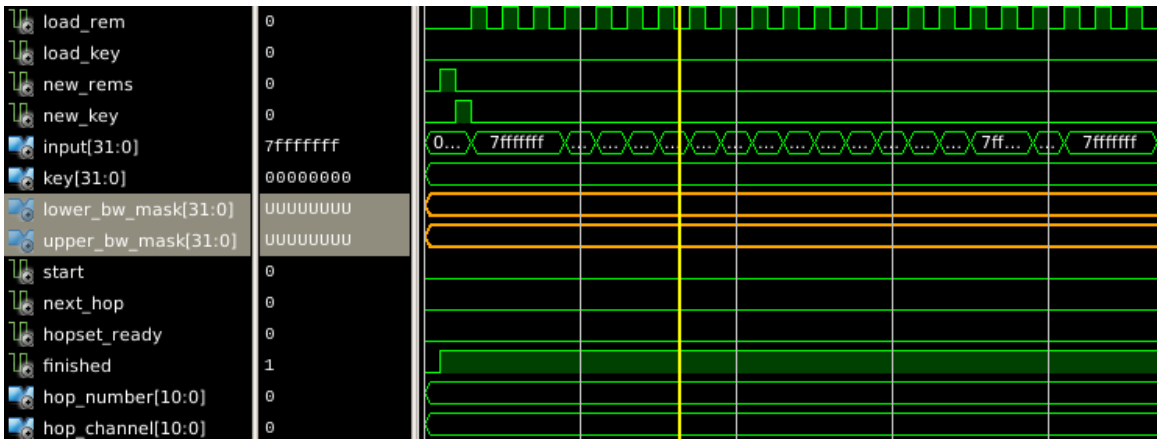


Figure 4.8: REM loading

The key is loaded in a similar fashion to the REM(s) with the `load_key` flag being toggled (Figure 4.9).

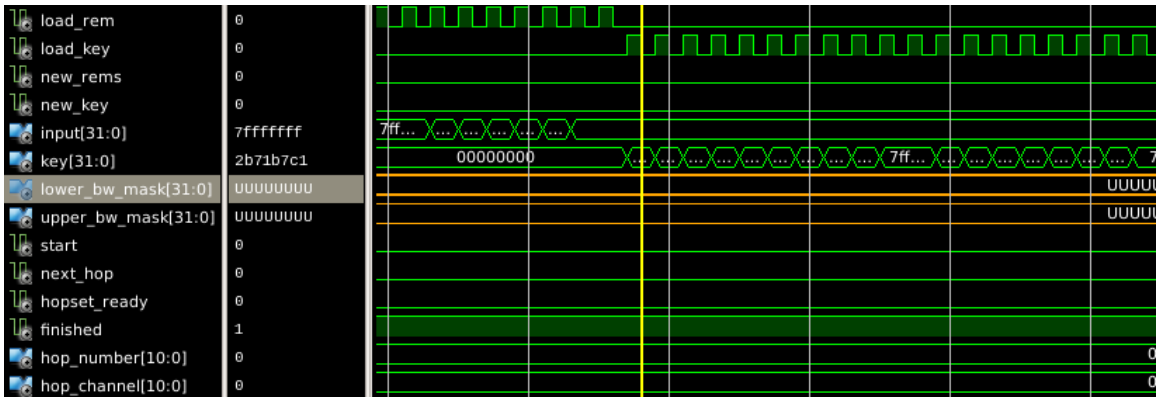


Figure 4.9: Key loading

Once the software has loaded the REMs and key, it toggles the `start` flag. The circuit immediately begins counting and tracking the open channels by first deasserting the `finished` flag (Figure 4.10).

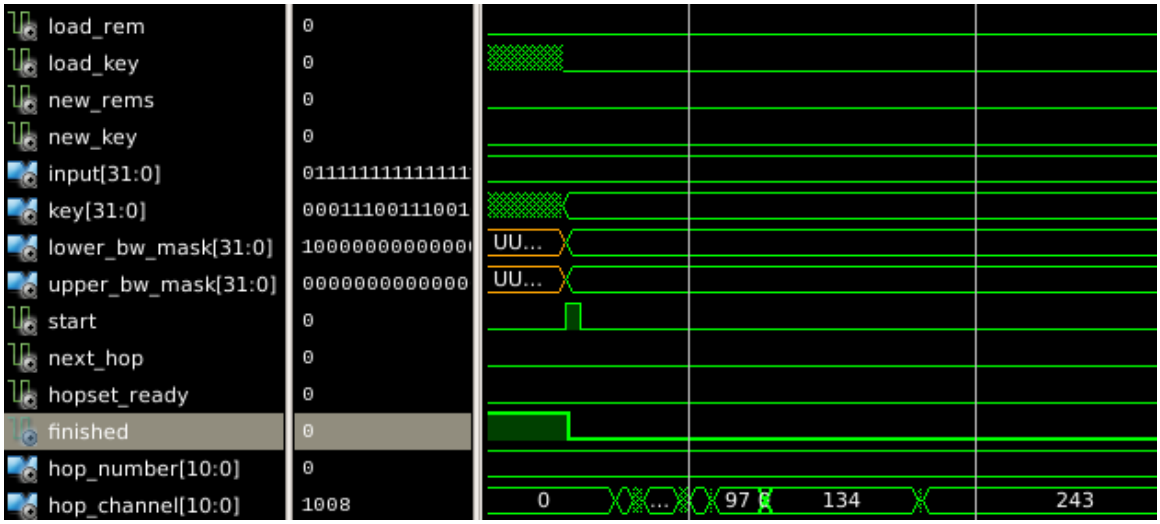


Figure 4.10: Channel counting

After the open channels have been counted, the `hopset_ready` flag is asserted. The software then toggles the `next_hop` flag in order to retrieve the hopset contents (Figure 4.11).

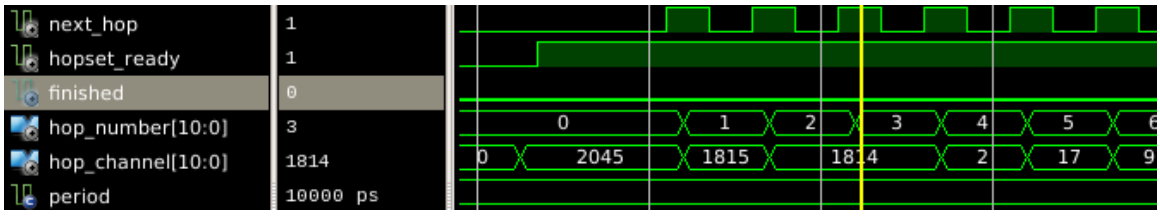


Figure 4.11: Hopset retrieval

The finished flag is asserted after all hops have been extracted. The `hopset_ready` flag is deasserted (Figure 4.12).

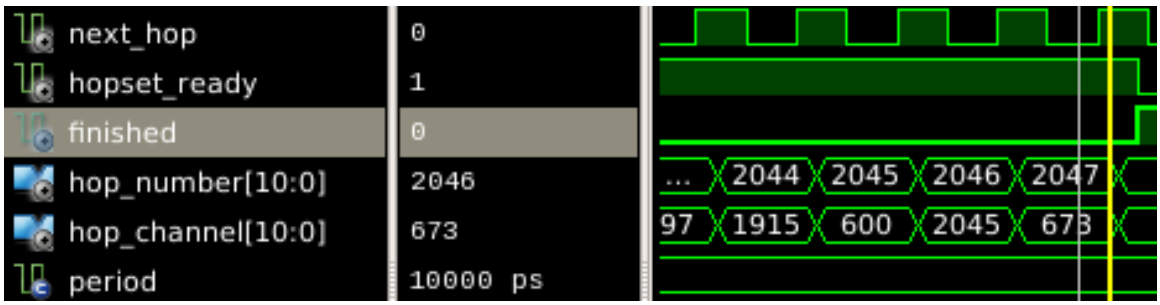


Figure 4.12: Hopset generation finished

Figure 4.13 shows full system simulation. Hopset output occupies the most time because its speed is governed by that of the software running on the processor.

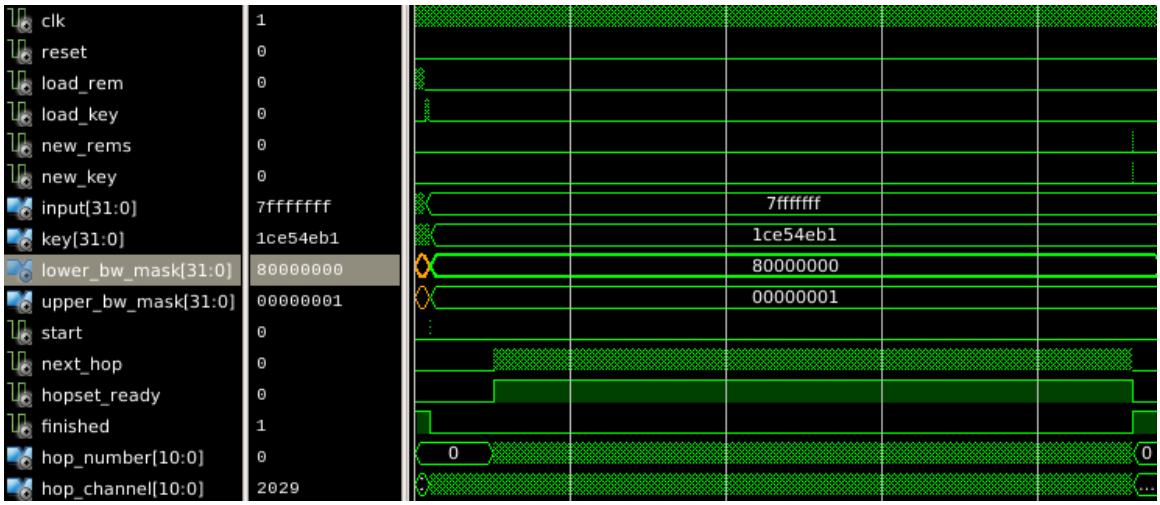


Figure 4.13: Full operation

4.2.2 Standalone Device Usage.

When placed and routed, the AHS (including a bus communication wrapper) occupies approximately 10% of the board's slice resources. Because BRAM is used, the associated slice usage is limited to a small footprint. Two BRAMs out of 376 total units are used to create the open channels table. Table 4.1 provides a comparison between device usage summaries for both the individual IP core and the WARP-based system as a whole.

4.2.3 Timing Analysis.

While the maximum frequency given by XST is only 70.442 MHz, the actual system can be run at a much higher frequency because of the automatic placement of clock dividers around the chip. For example, we run the PowerPC clock at 125 MHz and the bus clock at 100 MHz with no adverse effects. This is because the relatively low frequency applies only to a subset of clock nets.

Table 4.1: Device Resource Usage Summary.

Resource	Available	AHS	AHS+WARP
Number of BUFGs	32	1 (3%)	11 (34%)
Number of External IOBs	768	256 (33%)	462 (69%)
Number of RAMB16s	376	2 (1%)	59 (15%)
Number of Slices	42,176	4,614 (10%)	15,032 (35%)
Number of SLICEMs	21,088	768 (3%)	1,119 (5%)

4.2.4 Hopset Selection Demonstration.

In addition to actually building the AHS system and implementing on an FPGA, C code was written for testing and verifying the system's functionality. Figure 4.14 shows the output of the test. The example shown is the result of a 64-bit implementation. The smaller size (as opposed to the 2,048-bit full implementation) is used simply to aid in demonstration. Using all 2,048 bits would simply create a large, complicated output. The

```
AFIT Cognitive RAdio (ACRA) starting...
RF sensing not yet implemented.
Map sharing not yet implemented.
Clustering not yet implemented.
Printing hopset...
         0 25 50 37 10 21 43 23
        46 29 58 52 41 19 39 14
        29 59 54 45 27 55 47 31
        63 63 62 60 56 48 33  2
         4  8 16 33  2  4  9 19
        38 12 24 48 33  2  5 10
        21 42 20 41 19 38 13 27
        55 46 28 56 48 33  3  6
Done.
Hopset sending not yet implemented.
Hopset usage not yet implemented.
```

Figure 4.14: Hopset output example.

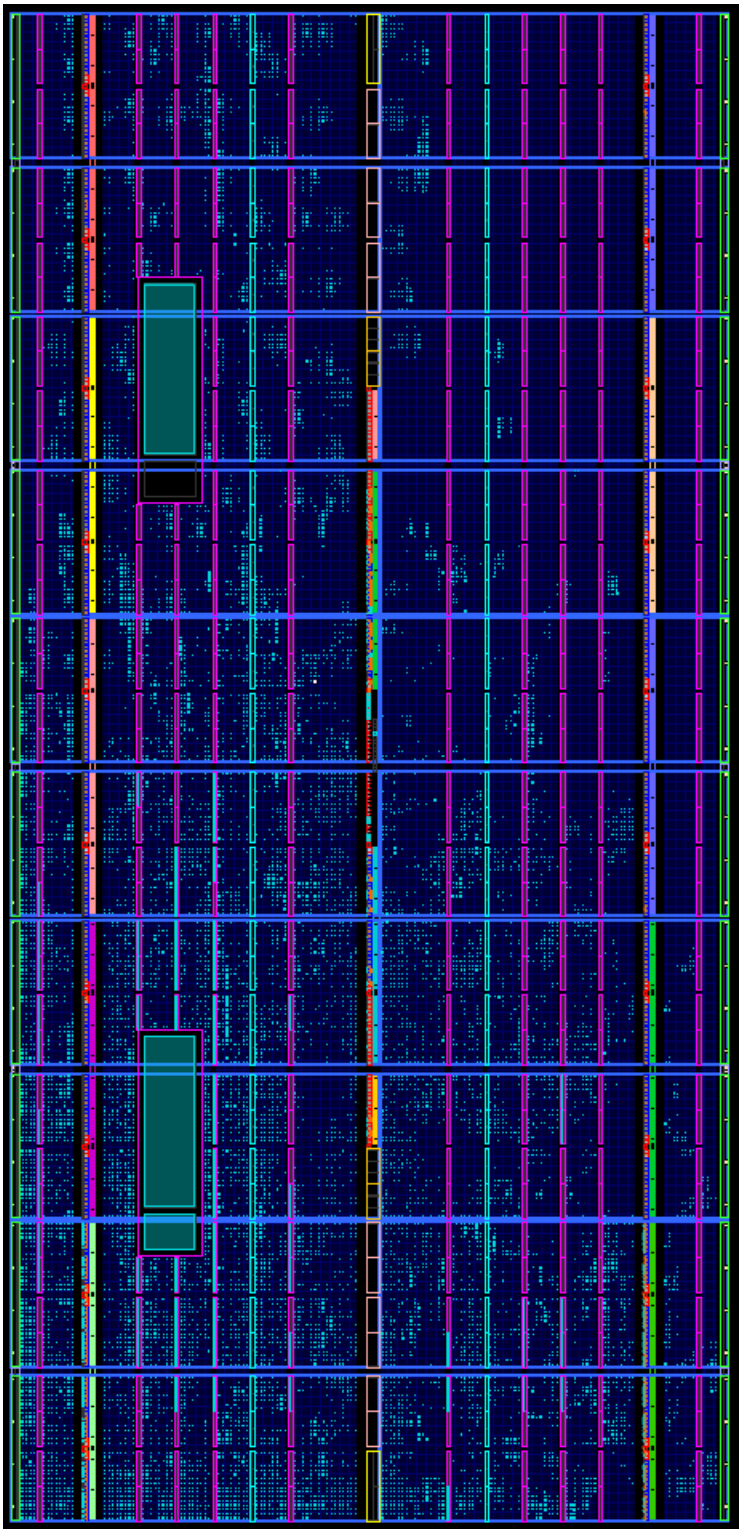


Figure 4.15: AHS+WARP device usage diagram.

hopset shown is the result of a 64-bit random key. The code used to produce this output forms the skeleton onto which the rest of the system is to be constructed.

4.2.5 Device Usage Floorplans.

The PlanAhead diagrams in Figures 4.15 and C.1 show how the slices (shown as light blue dots) fill the FPGA fabric with the existing WARP system and the AHS IP core and IP core alone, respectively. The most notable difference is the addition of routed PowerPC cores in the larger design (shown as two dim blue blocks).

4.2.6 Optimization Achievements.

Directing the synthesis and implementation tools to build the memory structure out of logic slices ran and did not complete, even when given over 12 hours to run. Purposefully using BRAM and using the XST option for automatically extracting BRAM was quick and resulted in virtually no synthesis and implementation overhead.

5 Conclusions

5.1 Research Contributions

This research makes five contributions to the field of cognitive radio:

1. The framework for a new hybrid hardware/software middleware architecture (see Figure 3.1);

2. A framework for testing and evaluating clustering algorithms in the context of cognitive radio networks (see Figure 3.1, **O-2** and Figure 3.5);

3. A new RF spectrum map representation technique (see Figure 3.6);

4. A new RF spectrum map merging technique (see Figure 3.1, **O-4** and see Figure 3.22);

5. A new method for generating a random, key-based adaptive hopset frequency hopping waveform (see Figure 3.1, **O-5** and see Figure 3.23); and

6. Initial integration testing toward implementing the proposed system on a field-programmable gate array (FPGA) (see Figure 4.14).

5.2 Whole System

We believe it is both possible and feasible to implement an adaptive frequency hopping cognitive radio for use among existing primary and secondary users. To support this assertion, we propose a new middleware architecture for use in such a system. We also implement several components as the basis for proving feasibility. Through defining the architecture and implementing several operations within the system, we conclude such a system is both possible and feasible to build.

5.3 Network Clustering

We presented a methodology for evaluating the k-means clustering algorithm in a cognitive radio network over a range of node distributions, cluster counts, and RF spectrum maps. We also introduced a new metric, intra-cluster spectrum similarity (ICSS), for comparing the effectiveness of clustering in the context of a dynamic RF spectrum environment. Our experiment showed that ICSS asymptotically approaches unity as the number of clusters in a notional cognitive radio network of 100 nodes approaches the number of nodes. This trend was as expected and remained consistent across all ten RF spectrum maps, distributions, distribution types, and clustering heuristics for a range of cluster counts. Because of its consistent performance, Lloyd's algorithm (the *de facto* heuristic) was selected for integration with the whole system.

In addition to selecting an algorithm, the consistency of results allows the design to incorporate the ICSS results in a table-lookup format. For example, if the radio determines that spectra must be Y-percent similar, the radio runs the clustering algorithm using X clusters. Given the requirement for Y, X is easily derived from the plotted ICSS data. Further, if it is determined a cluster must have high bandwidth, then a higher ICSS value (X) is chosen, and therefore a large number of clusters (Y) is chosen. Likewise, a low-bandwidth connection requires a lower ICSS value and therefore fewer clusters. Within this decision, there exists a tradeoff. High bandwidth leads to more clusters which leads to decreased complexity (toward $O(n\ log(n))$, whereas low bandwidth leads to fewer clusters which leads to increased complexity (toward $O(n)$). We conclude that the decision for Y is contingent upon a tradeoff between ICSS (representing similarity between REMs) and required bandwidth, and that the number of clusters is not dependent on ICSS alone, but also network requirements.

5.4 Adaptive Hopset Selection

The last three contributions were made in adaptive hopset selection. First, we implemented a cognitive radio with adaptive hopset selection on an FPGA. Second, we proposed and demonstrated a new technique for merging RF spectrum maps when represented as binary vectors. Finally, we proposed and demonstrated a new adaptive hopset selection technique. The whole system (composed of the WARP architecture and our custom core) fit within the resources available on the Virtex IV FPGA. Because our code simulation, mapped hardware simulation, and software-based testing yielded the desired results, this portion of the whole system functioned as expected.

5.5 Final Remarks

The architecture proposed in this research is a novel method by which to implement a frequency hopping cognitive radio network for coexistence with other RF spectrum users. By demonstrating the practicality of using the k-means clustering algorithm and RF spectrum measurements as an effective method for partitioning an otherwise-flat network into sub-networks, we provide an efficient way to decrease network complexity in a way that accounts for RF spectrum differences. Further, by demonstrating a hardware-based implementation for selecting a random frequency hopping hopset, we introduce an avenue for adaptively hopping frequencies as a means to quickly and continually avoid RF interference. To support these two contributions, we propose a lightweight, hardware-portable spectrum representation technique. It is expected these contributions forge a path for building on the proposed architecture and ultimately developing, testing, and fielding an operational frequency hopping cognitive radio network.

6 Future Work

As stated in the Introduction, several pieces within this research remain to be implemented before a full-scale, standalone prototype network can exist. At the time of publication, one post-doctoral student, two masters students, and several interns are currently working on expanding the envelope of this research.

1. **Multicast communication layer.** Multicast communication, specifically totally-ordered multicast (TOMC), enables a conceivably large and complex network to communicate in a structured fashion. Integrating the TOMC layer onto the board means tying in existing software. Currently, using the Spread API is yielding promising results. This layer will be fully validated once nodes can communicate over either wired or wireless links. This layer requires some sort of MAC layer to also be in place, so it is possible small and/or embedded operating system (i.e., TinyCore Linux, Windows Embedded, etc.) may need to run on the embedded processor. At time of print, this step is being completed as part of a follow-on student's thesis.

2. **Ethernet core.** An Ethernet core is the next step toward inter-node communication. When implemented, Ethernet allows two or more boards to be a part of the same wired network in much the same way as the eventual wireless network. At time of print, wireless transmission via Ethernet has been demonstrated but has not yet been integrated with the middleware architecture.

3. **Frequency hopping operation.** The WARP board supports up to four radio cards, and these cards must be integrated with the hopset selection method described in Section 3.3. Current work on this step (outside the scope of this document) is demonstrating the WARP's wireless functionality, so integration with the proposed middleware architecture is pending. In order for the radio to operate as intended,

though, this task is extremely critical. In addition to implementing frequency hopping-spread spectrum communication, it is desirable to include non-contiguous OFDM where carriers are adaptively selected using REM data.

4. **Secure hash chaining.** To realistically generate new hopsets periodically, keys need to be distributed securely and quickly. Secure hash chaining provides a method by which such values can be distributed. This task is intended for software-only implementation, but it is feasible the operation could be completed much quicker in a hardware-based FPGA solution.

5. **On-board spectrum sensing.** Once the board can sense its own spectrum and produce a model satisfying the previously requirements laid out and implemented, each board can begin to function as a fully autonomous device. Spectrum sensing is currently performed in an emulation environment; this is for prototyping only. Simply put, on-board sensing completes the OODA loop.

6. **Field testing.** Lab testing and emulation environments only validate this research to the accuracy of such tests. In order to fully prove this system's worth, it must be tested in an outdoor environment where the spectrum is truly unknown and the radio must truly adapt. This also allows the radio to grow as necessary without the physical limitations of doing so indoors.

Bibliography

[1] M. Faisal, Y. Park, and D. D. Wentzloff, "Reconfigurable Firmware-Defined Radios Synthesized from Standard Digital Logic Cells," 2011, pp. 803 115–803 115–8. [Online]. Available: http://dx.doi.org/10.1117/12.885058

[2] R. Coram, *Boyd: The Fighter Pilot Who Changed the Art of War*. Back Bay Books, 2002.

[3] B. Wang and K. Liu, "Advances in cognitive radio networks: A survey," in *IEEE Journal of Selected Topics in Signal Processing*, vol. 5, no. 1, February 2011, pp. 5–23.

[4] Air Force Chief Scientist, Office of the, *Report on Technology Horizons: A Vision for Air Force Science & Technology During 2010–2030*, 2010, pp. 52–55, 60, 78–91.

[5] D. Schneider, "LightSquared's GPS-Interference Controversy Comes to a Boil," in *IEEE Spectrum Magazine*, vol. 42, no. 2, February 2012, pp. 13–14.

[6] "Advanced RF Mapping (Radio Map)," *DARPA Strategic Technology Office*. [Online]. Available: http://www.darpa.mil/Our_Work/STO/Programs/Advanced_RF_Mapping_(Radio_Map).aspx

[7] "DARPA seeks mapping of RF spectrum for deployed troops," *Defense Systems*, March 2012. [Online]. Available: http://defensesystems.com/articles/2012/03/06/agg-darpa-spectrum-mapping.aspx

[8] M. McHenry, K. Steadman, A. Leu, and E. Melick, "XG DSA Radio System," in *3rd IEEE Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN)*, October 2008, pp. 1–11.

[9] F. Seelig, "A Description of the August 2006 XG Demonstrations at Fort A.P. Hill," in *2nd IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN)*, April 2007, pp. 1–12.

[10] B. Fette, Y. Zhao, B. Le, , and J. Reed, *Cognitive Radio Technology*, ser. Electronics & Electrical. Academic Press/Elsevier, 2009.

[11] SDR Forum, "SDRF Cognitive Radio Definitions," November 2007.

[12] D. W. Group, *IEEE Standard Definitions and Concepts for Dynamic Spectrum Access: Terminology Relating to Emerging Wireless Networks, System Functionality, and Spectrum Management*. Institute of Electrical and Electronics Engineers, October 2008.

[13] C. Cordeiro, K. Challapali, D. Birru, and N. Sai Shankar, "IEEE 802.22: The First Worldwide Wireless Standard Based on Cognitive Radios," in *First IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN)*, November 2005, pp. 328–337.

[14] S. Mellers, B. Richards, H.-H. So, S. Mishra, K. Camera, P. Subrahmanyam, and R. Brodersen, "Radio Testbeds Using BEE2," in *Signals, Systems, and Computers*, November 2007, pp. 1991–1995.

[15] B. Le, F. Rodriguez, Q. Chen, B. Li, F. Ge, M. ElNainay, T. Rondeau, and C. Bostian, "A Public Safety Cognitive Radio Node," in *SDR Forum Technical Conference*, 2007.

[16] A. Mody, M. Sherman, R. Martinez, R. Reddy, and T. Kiernan, "Survey of IEEE standards supporting cognitive radio and dynamic spectrum access," in *IEEE Military Communications Conference (MILCOM)*, November 2008, pp. 1–7.

[17] "Single Channel Ground and Airborne Radio System (SINCGARS)," 1999. [Online]. Available: http://www.fas.org/man/dod-101/sys/land/sincgars.htm

[18] "HAVE QUICK Frequency Hopping System," 2012. [Online]. Available: http://www.cryptomuseum.com/radio/havequick/index.htm

[19] "Single Channel Military Comm Recorder," 2012. [Online]. Available: http://www.digital-loggers.com/sincgars.html

[20] "UHF Radio Control Panel," 2002. [Online]. Available: http://www.xflight.de/pe_org_par_lfc_uhf.htm

[21] D. Stranneby and P. Kallquist, "Adaptive Frequency Hopping in HF Environments," in *IEEE Military Communications Conference (MILCOM)*, vol. 1, October 1993, pp. 338–341.

[22] D. Herrick, P. Lee, and J. Ledlow, L.L., "Correlated Frequency Hopping - An Improved Approach to HF Spread Spectrum Communications," in *Proceedings of the 1996 Tactical Communications Conference*, April 1996, pp. 319–324.

[23] K. Hamdi and O. Bamahdi, "A New Adaptive Frequency Hopping Technique," in *60th IEEE Vehicular Technology Conference*, vol. 3, September 2004, pp. 2083–2086.

[24] O. Bamahdi and S. Zummo, "An Adaptive Frequency Hopping Technique With Application to Bluetooth-WLAN Coexistence," in *International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies*, April 2006, p. 131.

[25] P. Popovski, H. Yomo, and R. Prasad, "Dynamic adaptive frequency hopping for mutually interfering wireless personal area networks," in *IEEE Transactions on Mobile Computing*, vol. 5, no. 8, August 2006, pp. 991–1003.

[26] S. Mishra, A. Sahai, and R. Brodersen, "Cooperative Sensing among Cognitive Radios," in *IEEE International Conference on Communications*, vol. 4, June 2006, pp. 1658–1663.

[27] C. Sun, W. Zhang, and K. Letaief, "Cooperative Spectrum Sensing for Cognitive Radios under Bandwidth Constraints," in *IEEE Wireless Communications and Networking Conference*, March 2007, pp. 1–5.

[28] C. Sun, W. Zhang, and K. Ben, "Cluster-Based Cooperative Spectrum Sensing in Cognitive Radio Systems," in *IEEE International Conference on Communications*, June 2007, pp. 2511–2515.

[29] Y. Zhao, J. H. Reed, S. Mao, and K. K. Bae, "Overhead Analysis for Radio Environment Mapenabled Cognitive Radio Networks," in *IEEE Workshop on Networking Technologies for Software Defined Radio Networks*, September 2006, pp. 18–25.

[30] X. Defago, A. Schiper, and P. Urban, "Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey," in *ACM Computing Surveys (CSUR)*, vol. 36, no. 4, December 2004, pp. 372–421.

[31] Y. Zhao, J. Gaeddert, L. Morales, K. Bae, J.-S. Um, and J. H. Reed, "Development of Radio Environment Map Enabled Case- and Knowledge-Based Learning Algorithms for IEEE 802.22 WRAN Cognitive Engines," in *2nd International Conference on Cognitive Radio Oriented Wireless Networks and Communications (CROWNCOM)*, August 2007, pp. 44–49.

[32] D. Denkovski, V. Atanasovski, L. Gavrilovska, J. Riihijarvi, and P. Mahonen, "Reliability of a radio environment Map: Case of spatial interpolation techniques," in *7th International ICST Conference on Cognitive Radio Oriented Wireless Networks and Communications (CROWNCOM)*, June 2012, pp. 248–253.

[33] L. Iacobelli, P. Fouillot, and C. Le Martret, "Radio Environment Map based architecture and protocols for mobile ad hoc networks," in *The 11th Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net)*, June 2012, pp. 32–38.

[34] D.-Y. Seol, H.-J. Lim, and G.-H. Im, "Optimal threshold adaptation with radio environment map for cognitive radio networks," in *IEEE International Symposium on Information Theory*, July 2009, pp. 2527–2531.

[35] F. Ge, R. Rangnekar, A. Radhakrishnan, S. Nair, Q. Chen, A. Fayez, Y. Wang, and C. Bostian, "A Cooperative Sensing Based Spectrum Broker for Dynamic Spectrum Access," in *Military Communications Conference (MILCOM)*, October 2009, pp. 1–7.

[36] T. Kanungo, D. Mount, N. Netanyahu, C. D. Piatko, R. Silverman, and A. Wu, "An Efficient k-Means Clustering Algorithm: Analysis and Implementation," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24.7, July 2002, pp. 881–892.

[37] D. Mount, "KMlocal: A Testbed for k-means Clustering Algorithms," Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland 20742, Tech. Rep., August 2005.

[38] G. Minden, J. Evans, L. Searl, D. DePardo, V. Petty, R. Rajbanshi, T. Newman, Q. Chen, F. Weidling, J. Guffey, D. Datla, B. Barker, M. Peck, B. Cordill, A. Wyglinski, and A. Agah, "KUAR: A Flexible Software-Defined Radio Development Platform," in *2nd IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN)*, April 2007, pp. 428–439.

[39] K. Amiri, Y. Sun, P. Murphy, C. Hunter, J. Cavallaro, and A. Sabharwal, "WARP, a Unified Wireless Network Testbed for Education and Research," in *IEEE*

International Conference on Microelectronic Systems Education, June 2007, pp. 53–54.

[40] J. Lotze, S. A. Fahmy, J. Noguera, L. Doyle, and R. Esser, "An FPGA-Based Cognitive Radio Framework," in *IET Irish Signals and Systems Conference*, June 2008, pp. 138–143.

[41] "XST User Guide," Xilinx, Inc., Tech. Rep., September 2009.

Appendix A: DYSE-Generated RF Spectrum Maps

The following plots represent "snapshots" of various RF spectrum environments. These plots were generated using AFRL's DYSE system. Dashed lines represent the threshold applied to each plot.

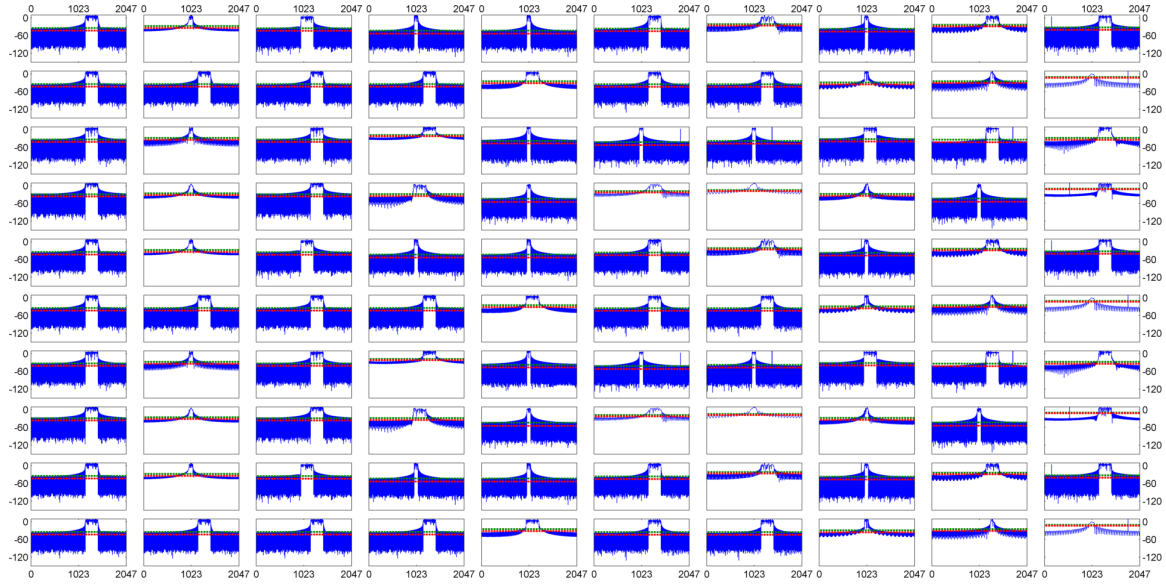


Figure A.1: DYSE-generated RF spectrum map #1.

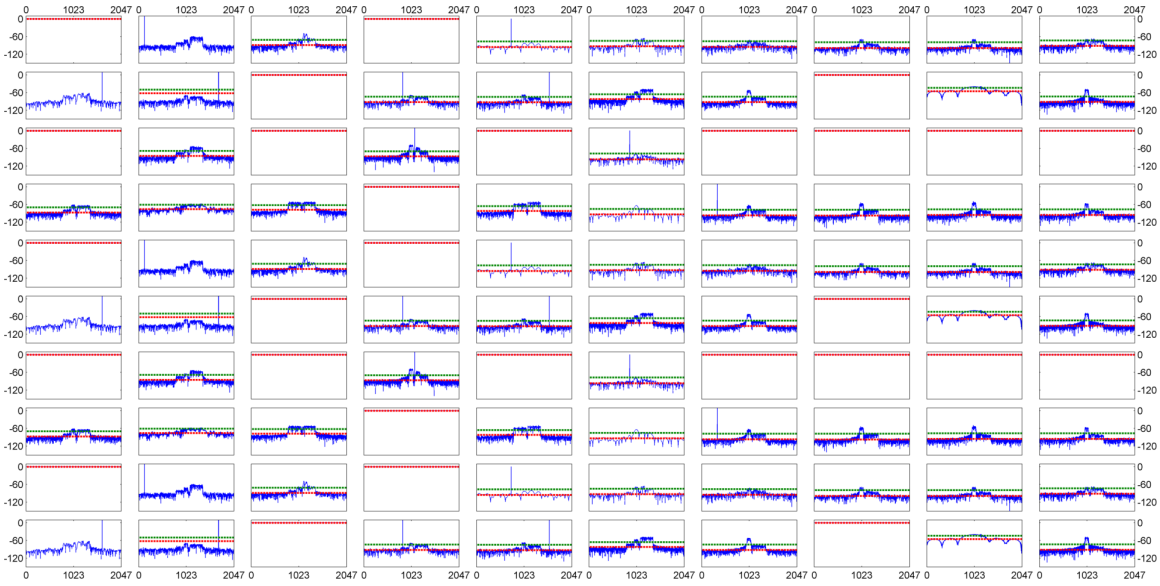


Figure A.2: DYSE-generated RF spectrum map #2.

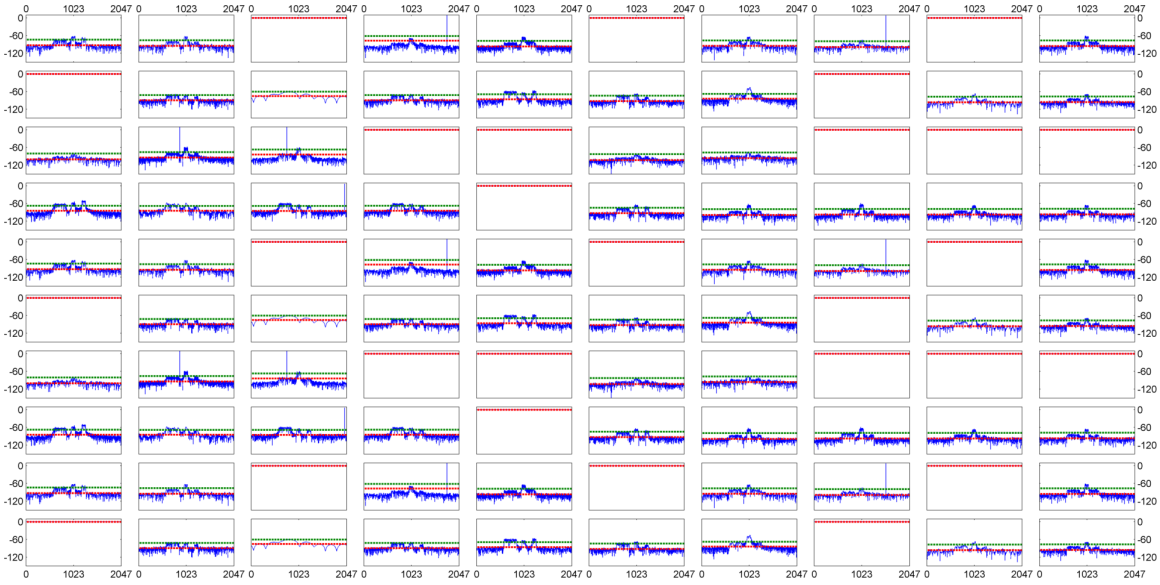


Figure A.3: DYSE-generated RF spectrum map #3.

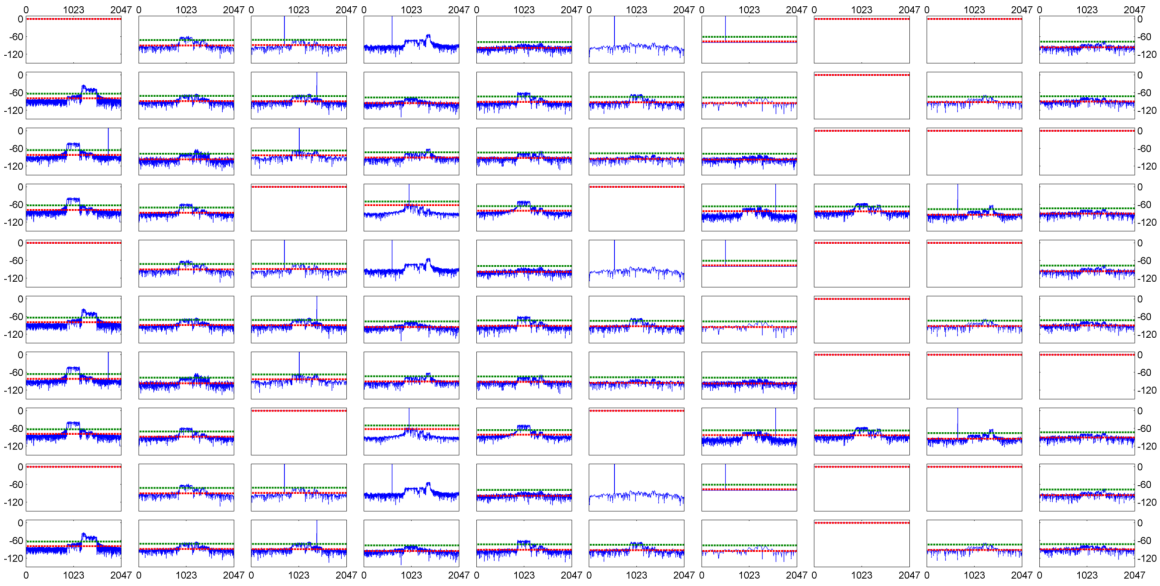


Figure A.4: DYSE-generated RF spectrum map #4.

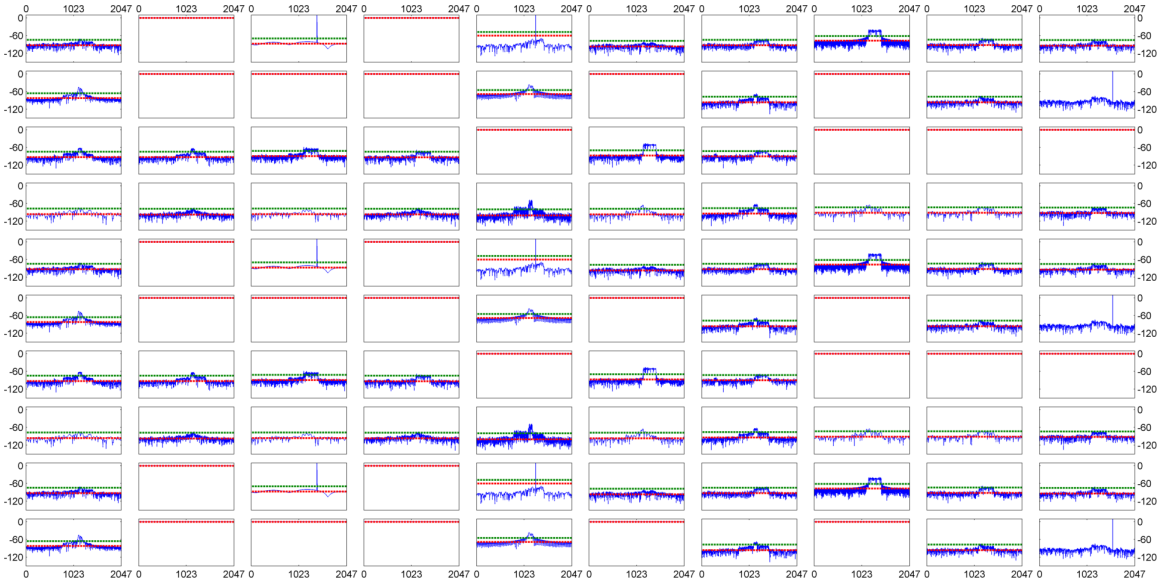


Figure A.5: DYSE-generated RF spectrum map #5.

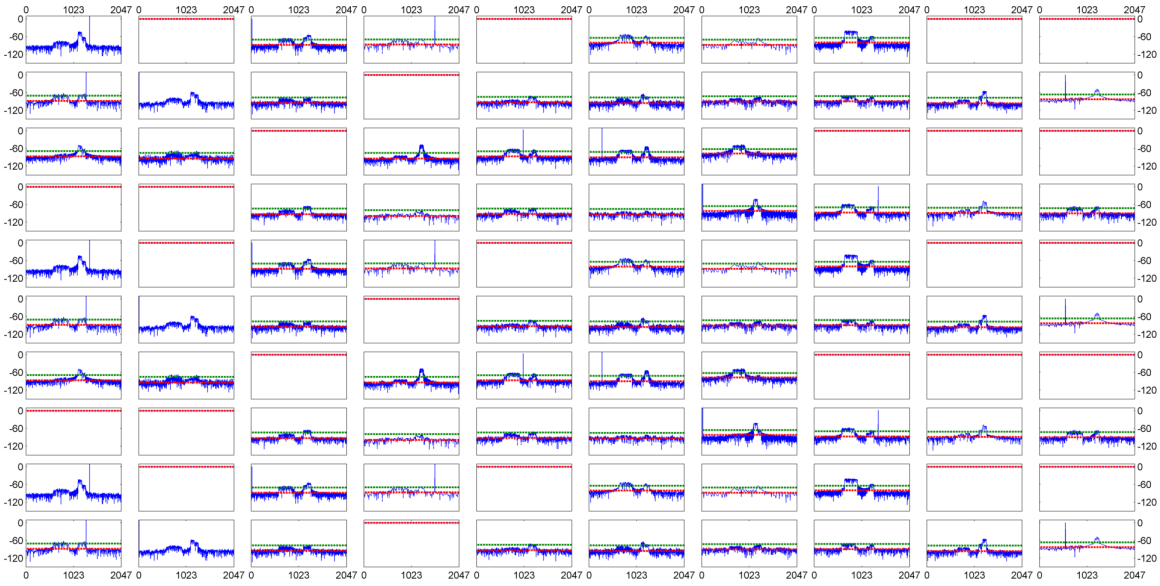


Figure A.6: DYSE-generated RF spectrum map #6.

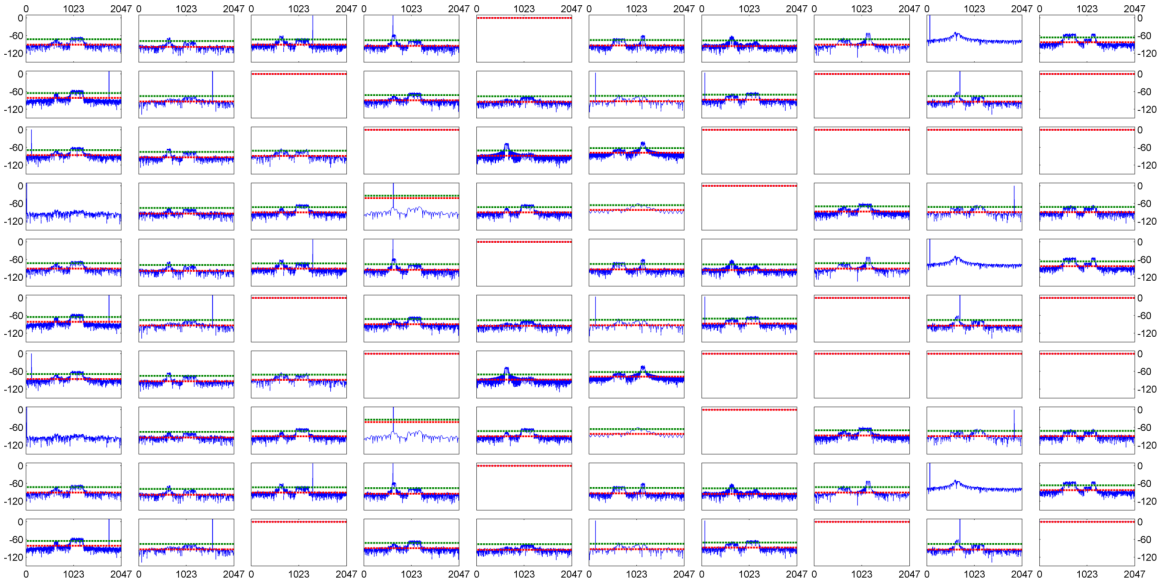


Figure A.7: DYSE-generated RF spectrum map #7.

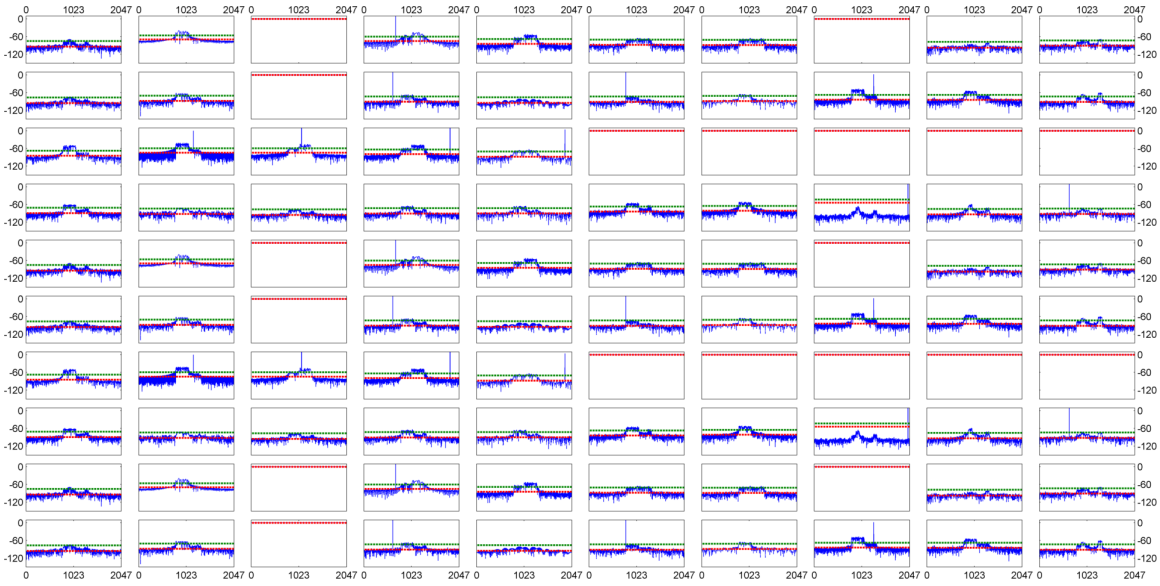


Figure A.8: DYSE-generated RF spectrum map #8.

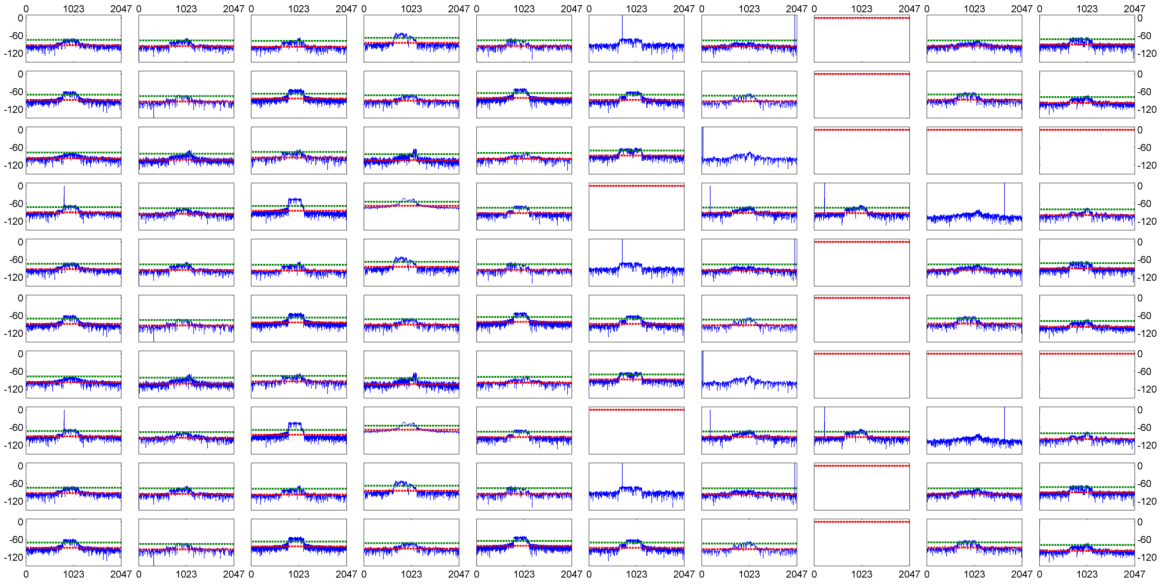


Figure A.9: DYSE-generated RF spectrum map #9.

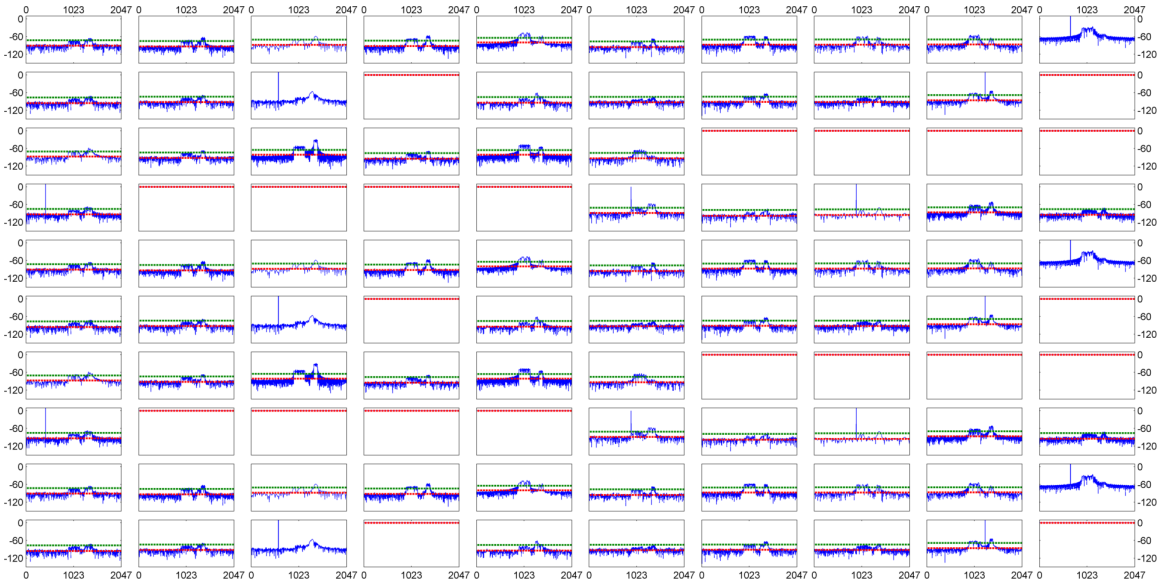


Figure A.10: DYSE-generated RF spectrum map #10.

Appendix B: Node Distributions

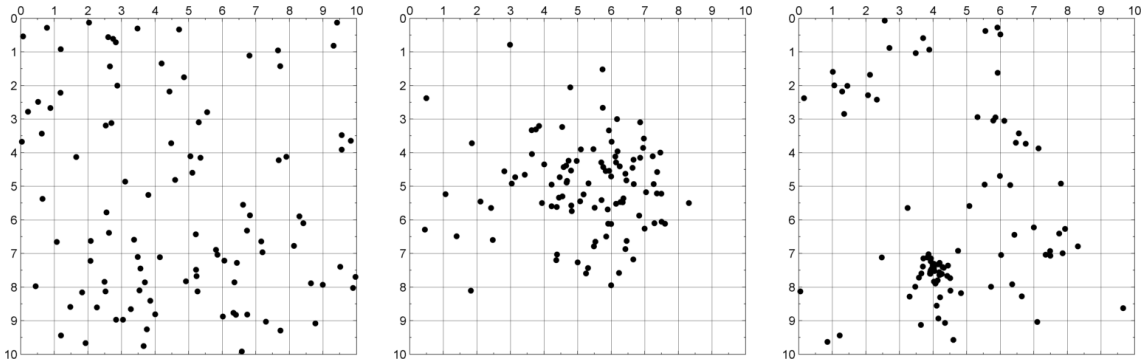


Figure B.1: Uniform (l), Gauss (c), and multi-cluster (r) distributions (seed = 1).

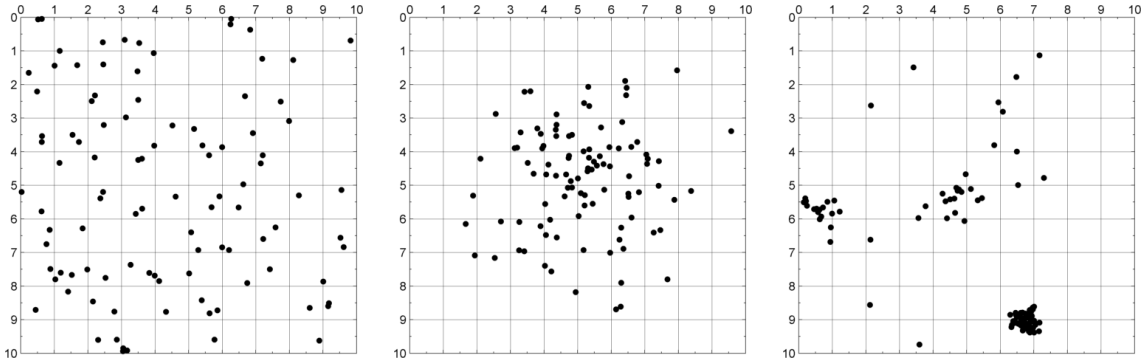


Figure B.2: Uniform (l), Gauss (c), and multi-cluster (r) distributions (seed = 2).

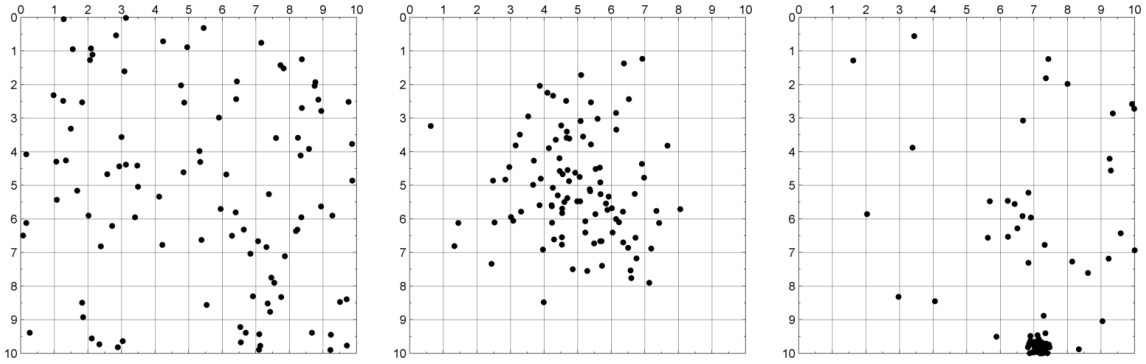


Figure B.3: Uniform (l), Gauss (c), and multi-cluster (r) distributions (seed = 4).

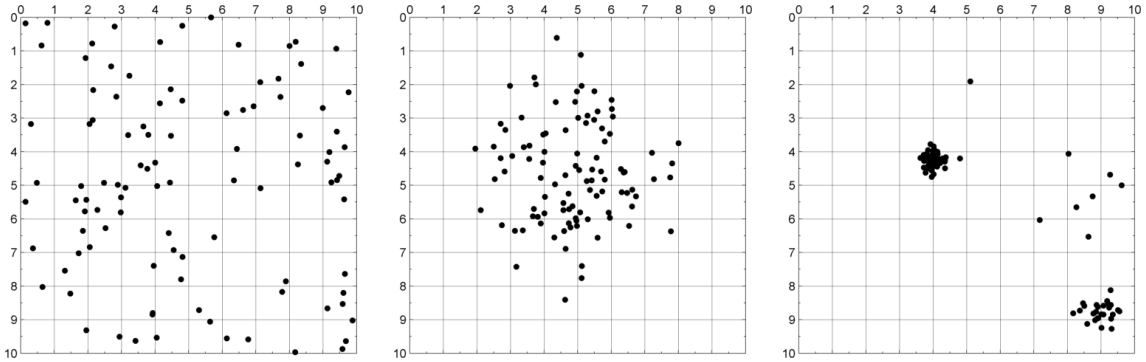


Figure B.4: Uniform (l), Gauss (c), and multi-cluster (r) distributions (seed = 5).

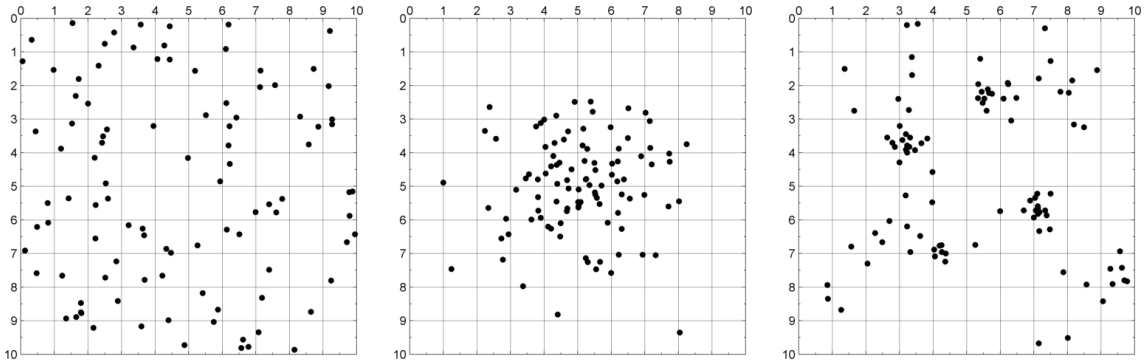


Figure B.5: Uniform (l), Gauss (c), and multi-cluster (r) distributions (seed = 6).

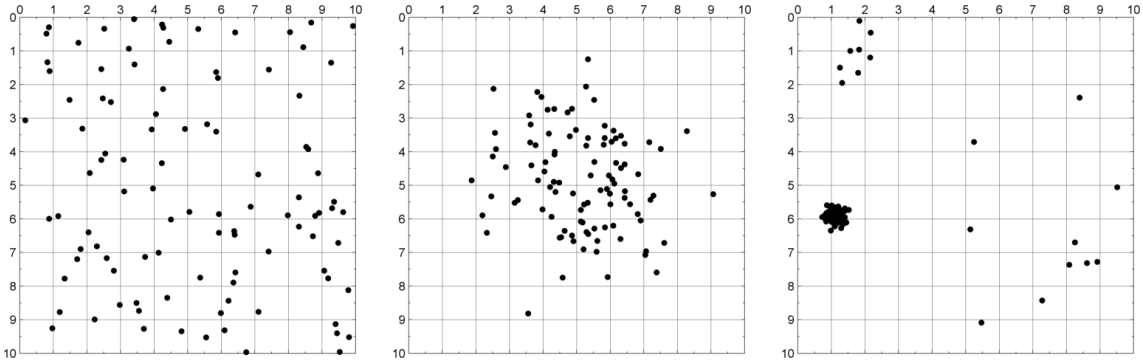


Figure B.6: Uniform (l), Gauss (c), and multi-cluster (r) distributions (seed = 7).

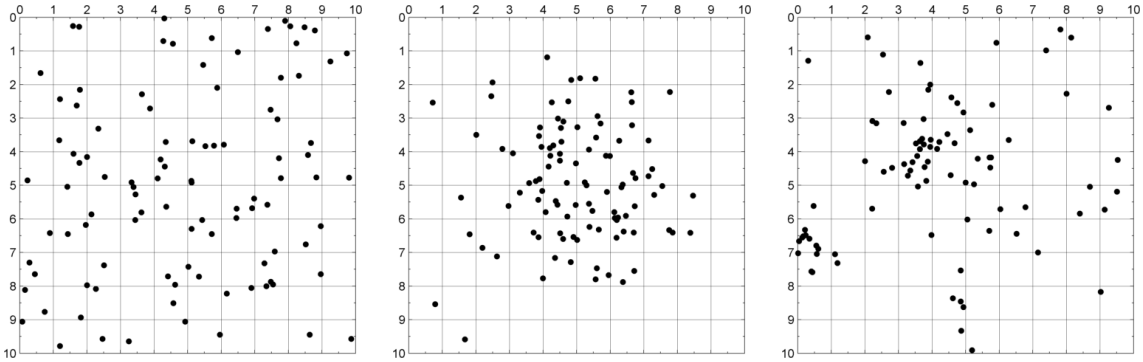


Figure B.7: Uniform (l), Gauss (c), and multi-cluster (r) distributions (seed = 8).

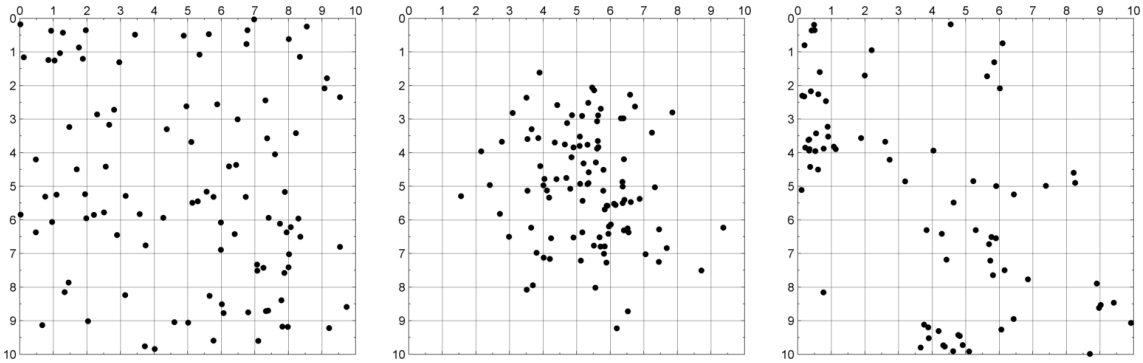


Figure B.8: Uniform (l), Gauss (c), and multi-cluster (r) distributions (seed = 9).

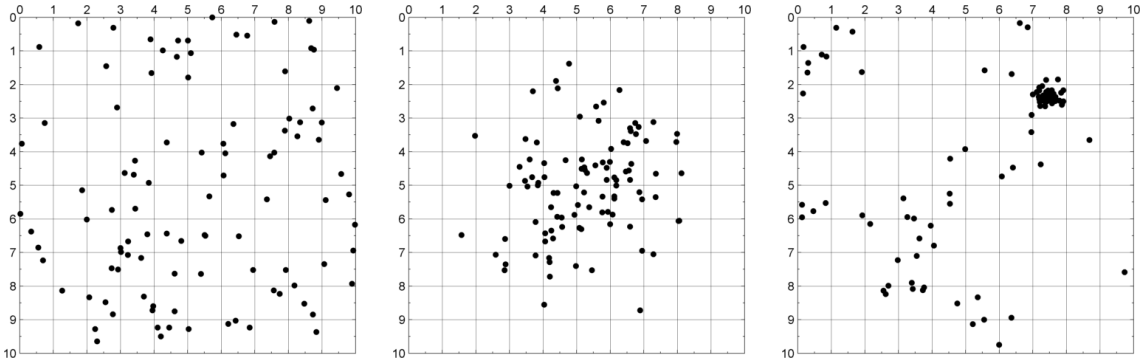


Figure B.9: Uniform (l), Gauss (c), and multi-cluster (r) distributions (seed = 10).

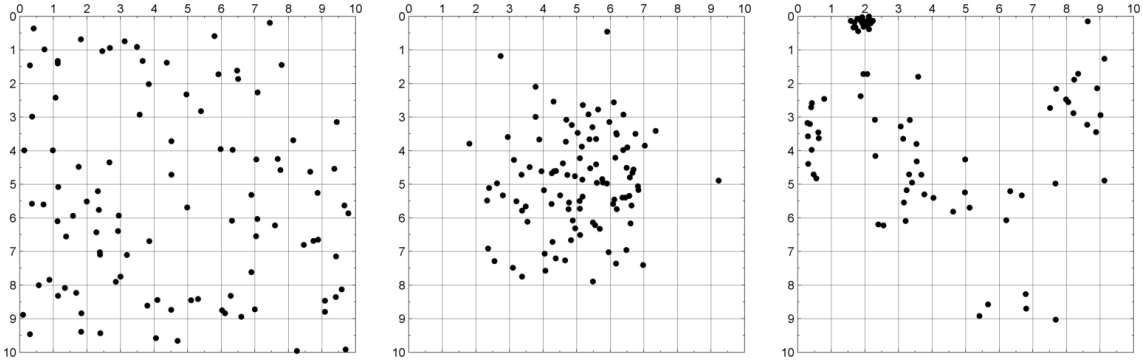


Figure B.10: Uniform (l), Gauss (c), and multi-cluster (r) distributions (seed = 11).

Appendix C: Additional FPGA Design Figures

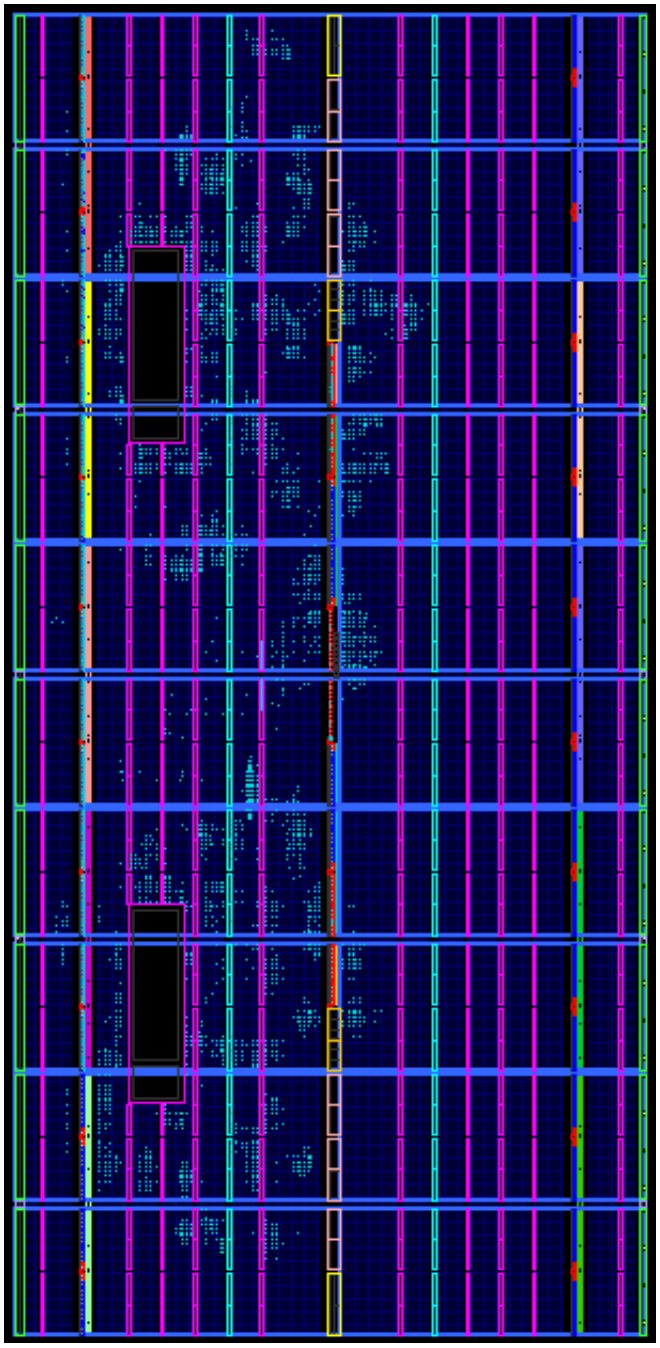


Figure C.1: AHS device usage diagram.

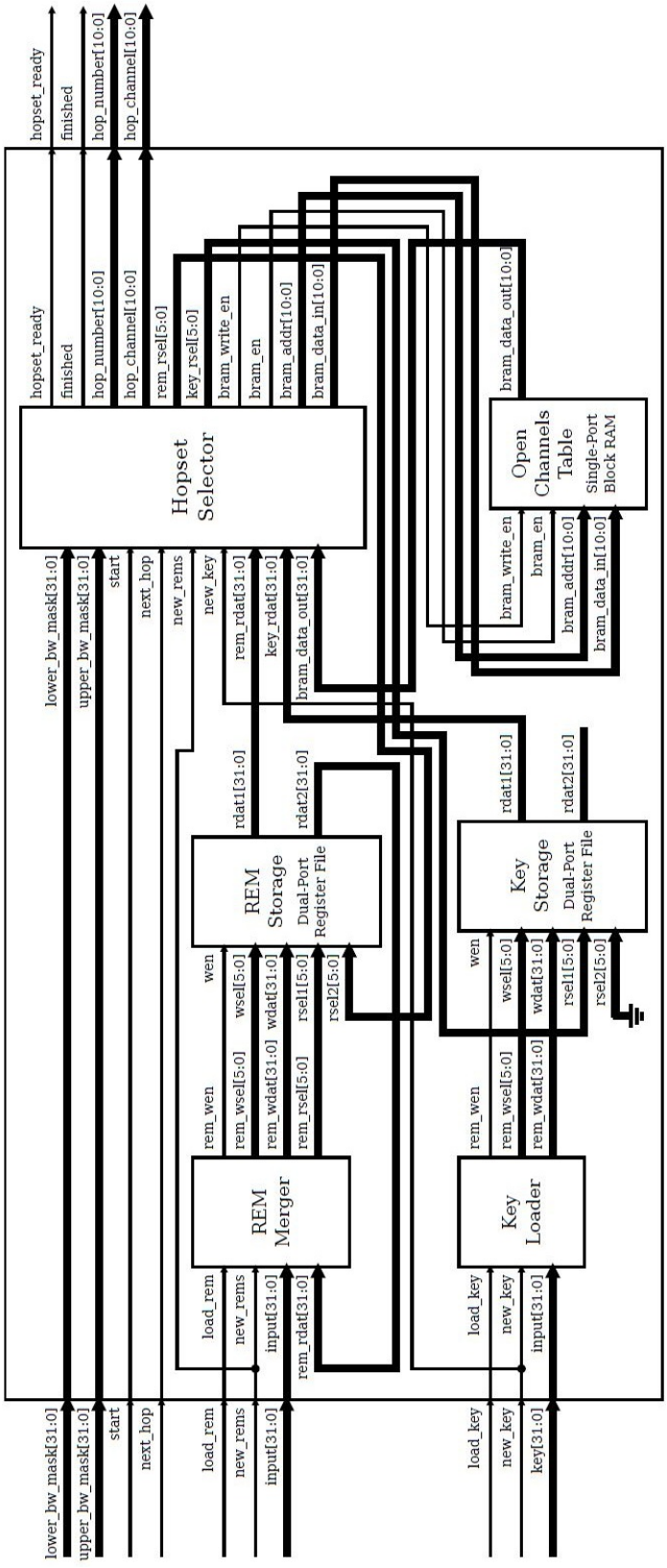


Figure C.2: AHS internal structure.

Appendix D: Additional Clustering Visualization Plots

Figure D.1: Baseline node distribution with two apparent clusters.

Figure D.2: Baseline node distribution with four apparent clusters.

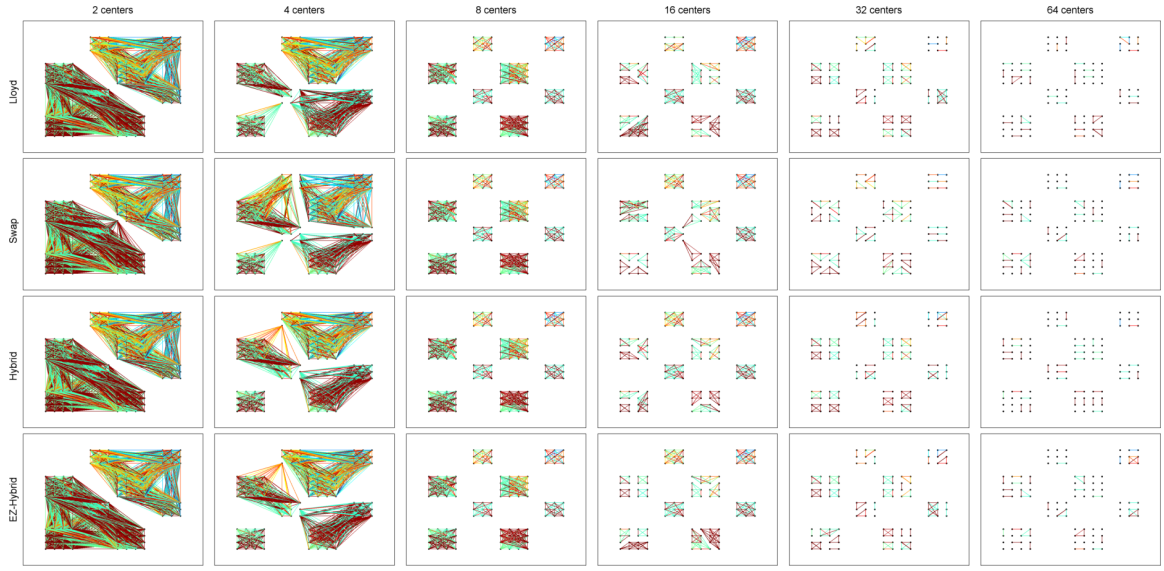


Figure D.3: Baseline node distribution with eight apparent clusters.

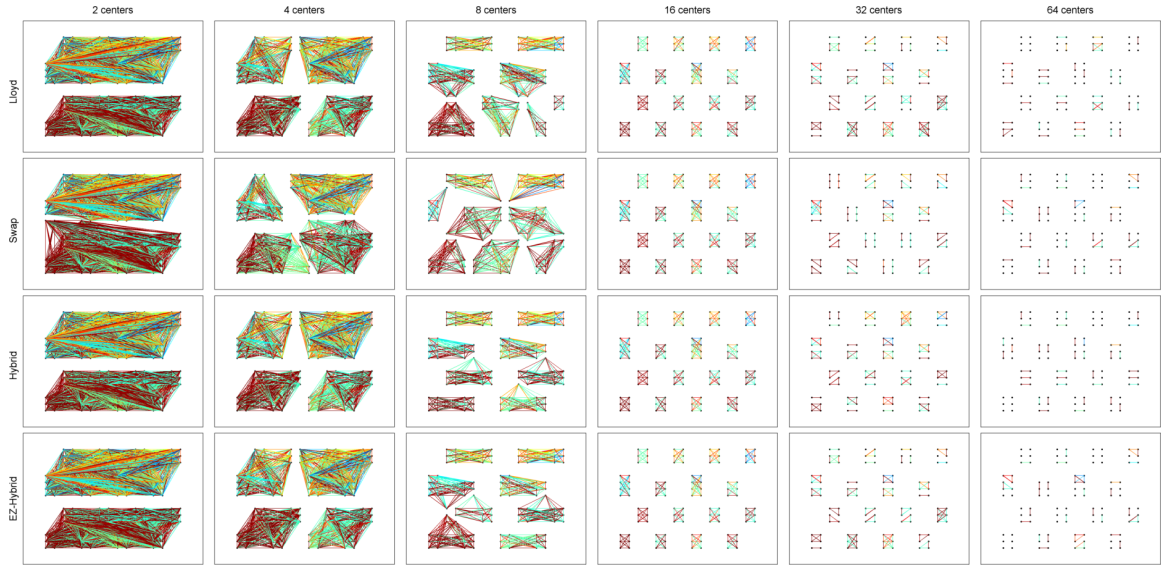


Figure D.4: Baseline node distribution with 16 apparent clusters.

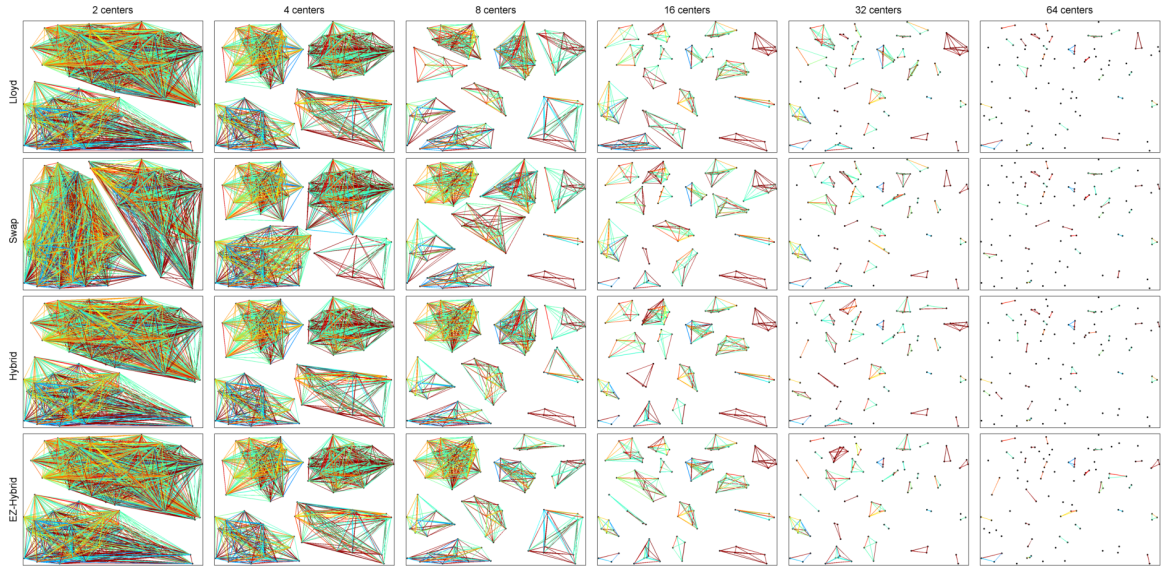


Figure D.5: Uniform node distribution using DYSE map #1.

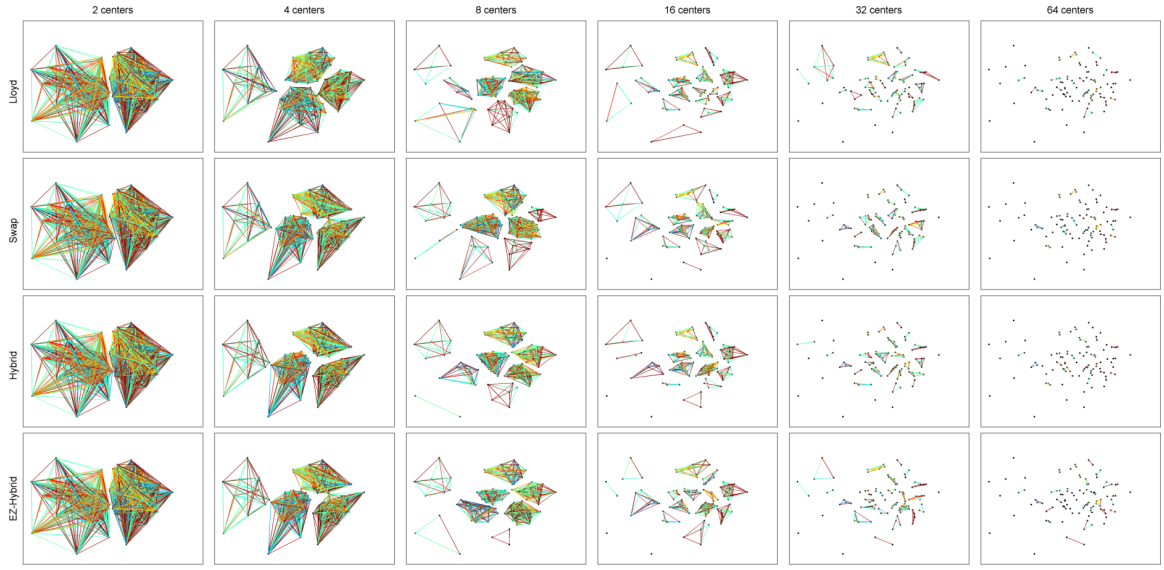


Figure D.6: Gauss node distribution using DYSE map #1.

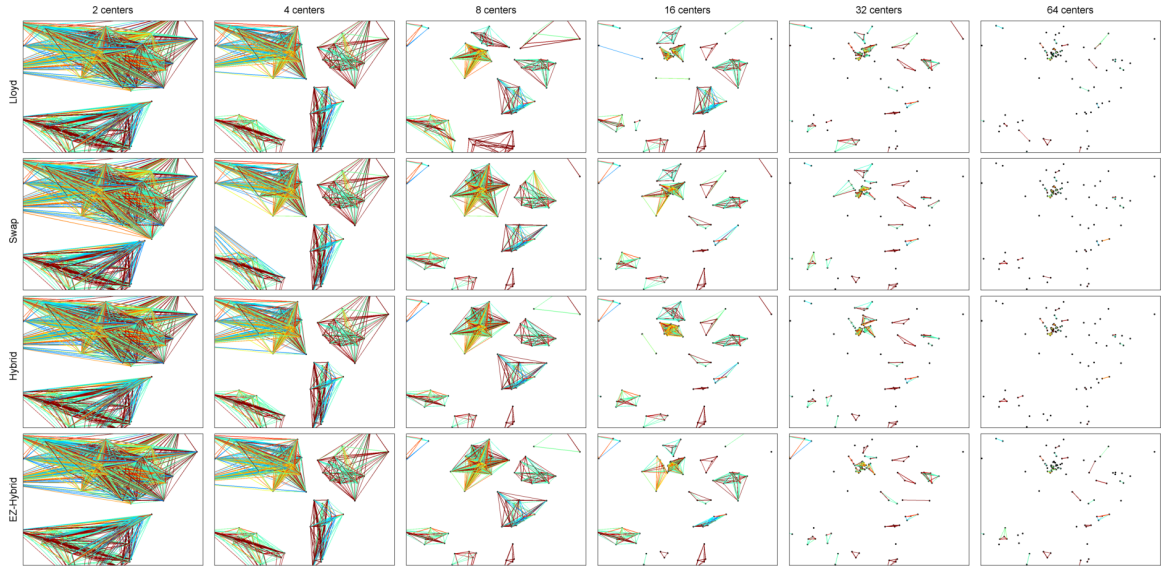


Figure D.7: Multi-cluster node distribution using DYSE map #1.

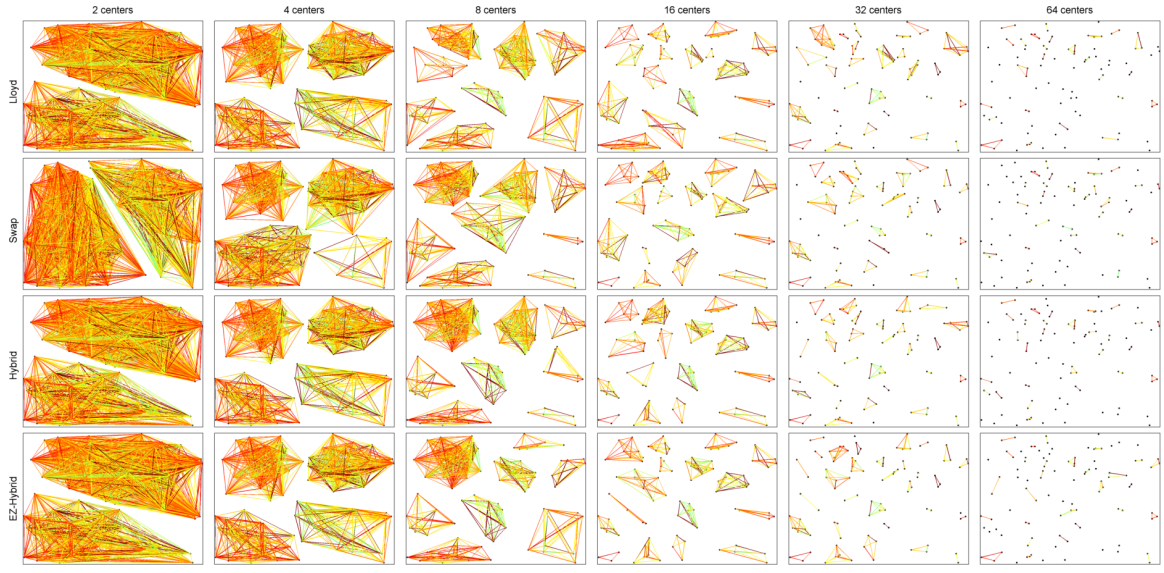


Figure D.8: Uniform node distribution using DYSE map #2.

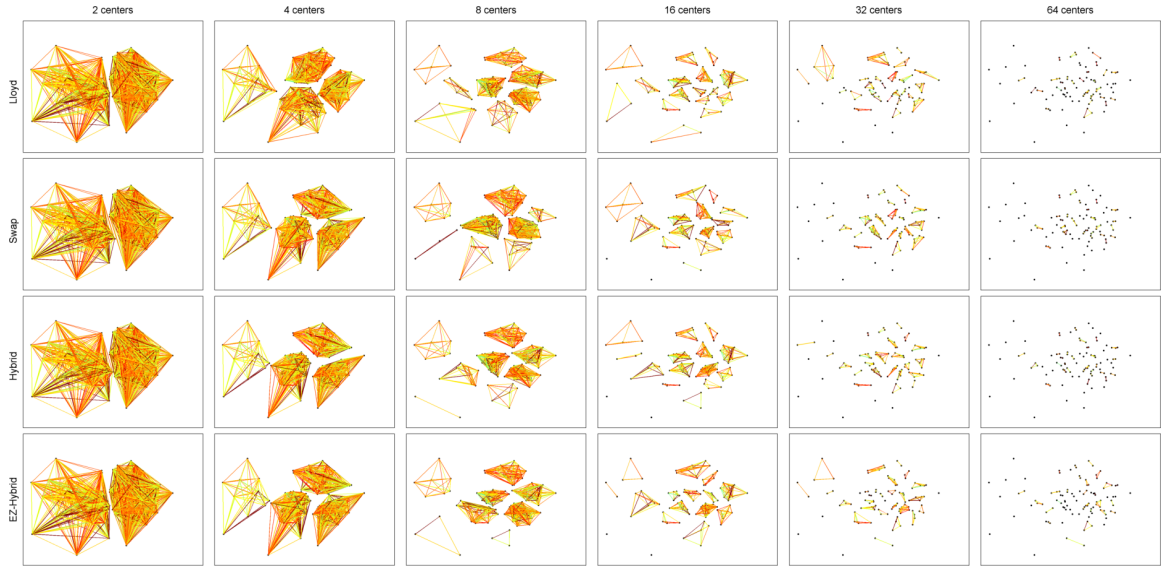


Figure D.9: Gauss node distribution using DYSE map #2.

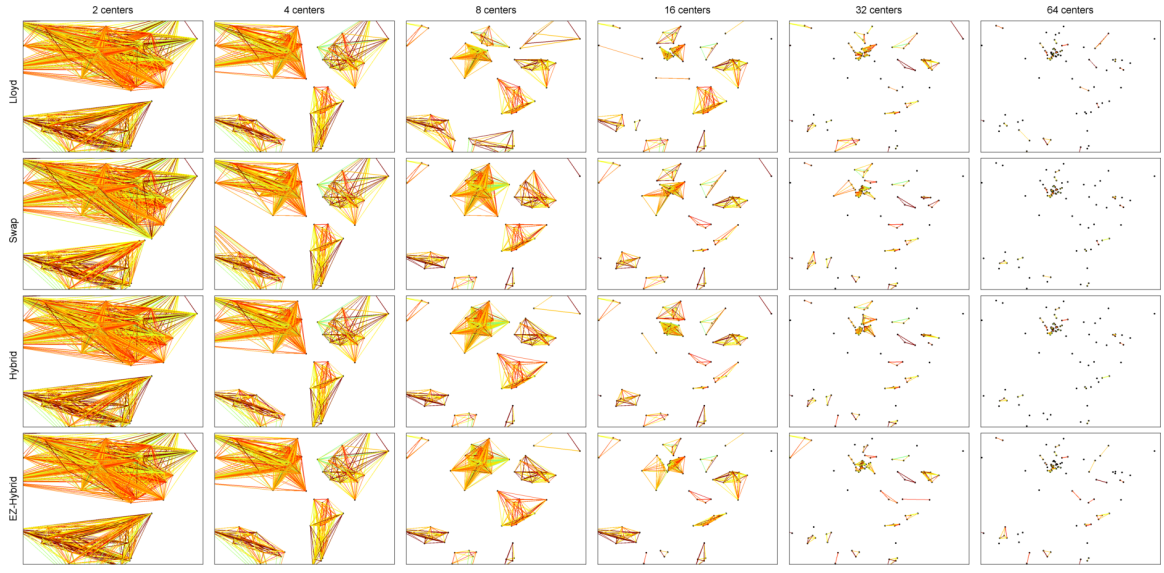


Figure D.10: Multi-cluster node distribution using DYSE map #2.

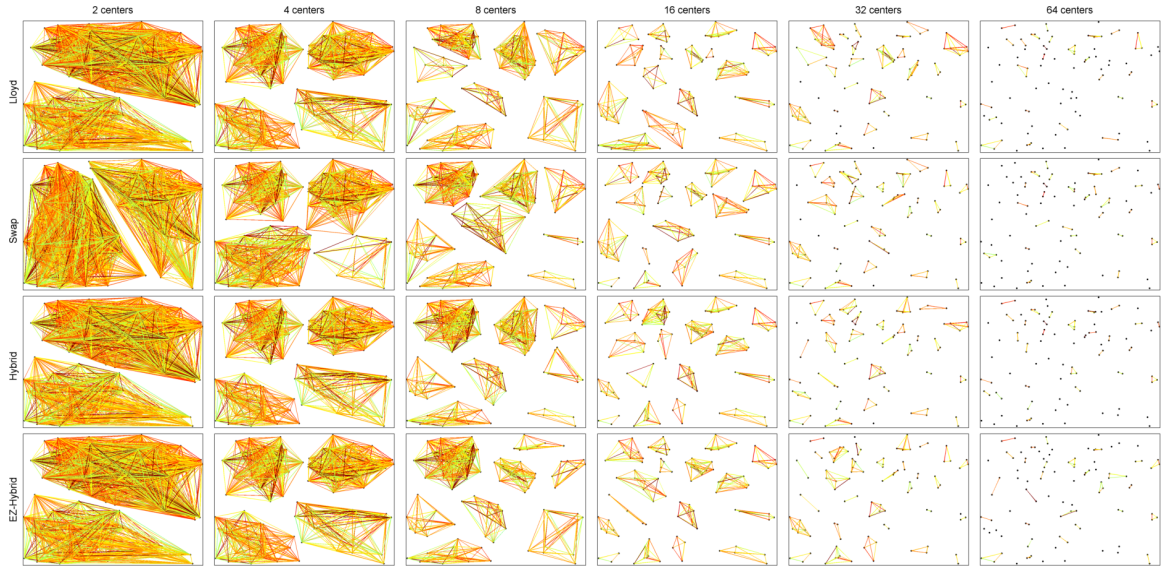


Figure D.11: Uniform node distribution using DYSE map #3.

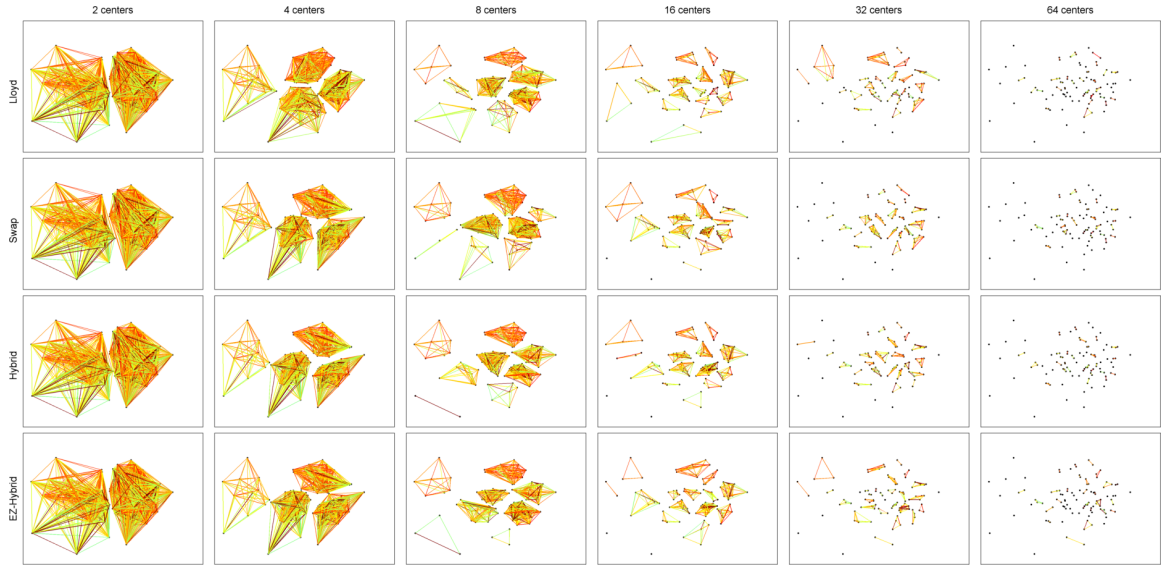


Figure D.12: Gauss node distribution using DYSE map #3.

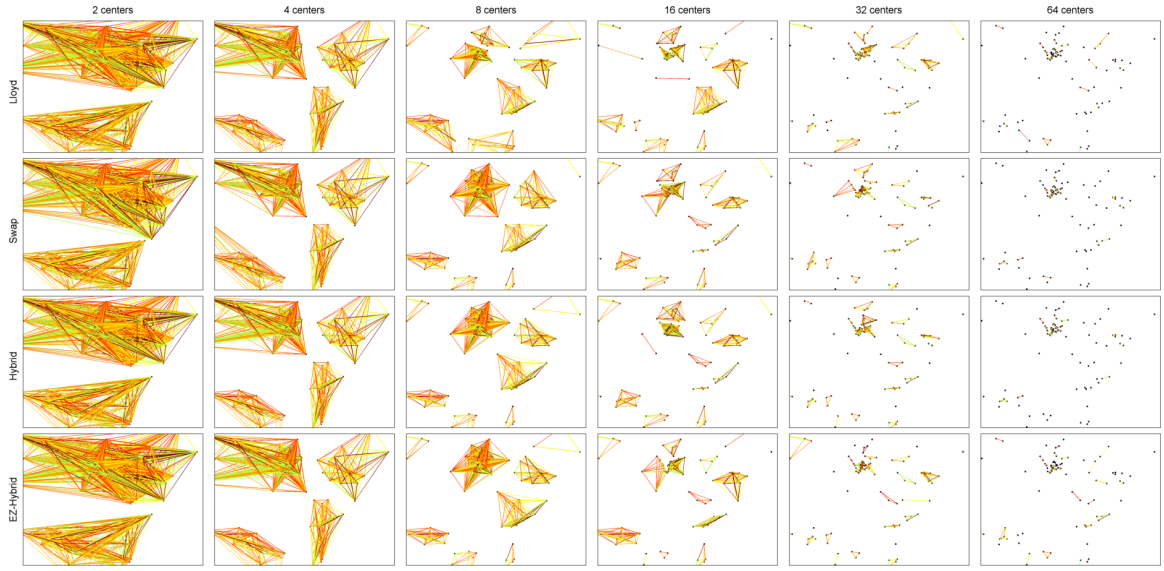


Figure D.13: Multi-cluster node distribution using DYSE map #3.

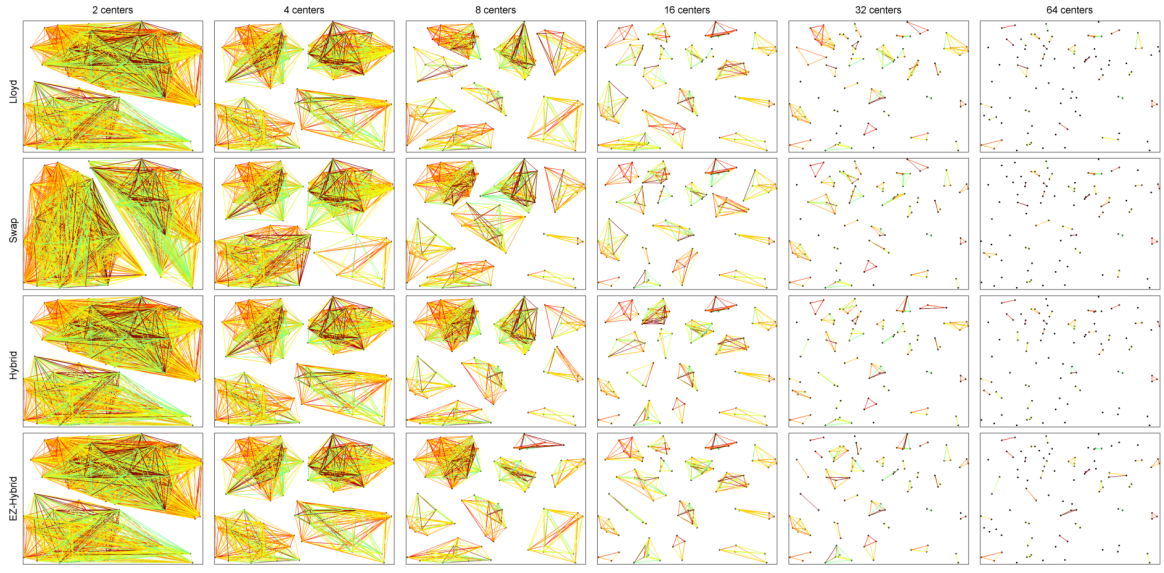


Figure D.14: Uniform node distribution using DYSE map #4.

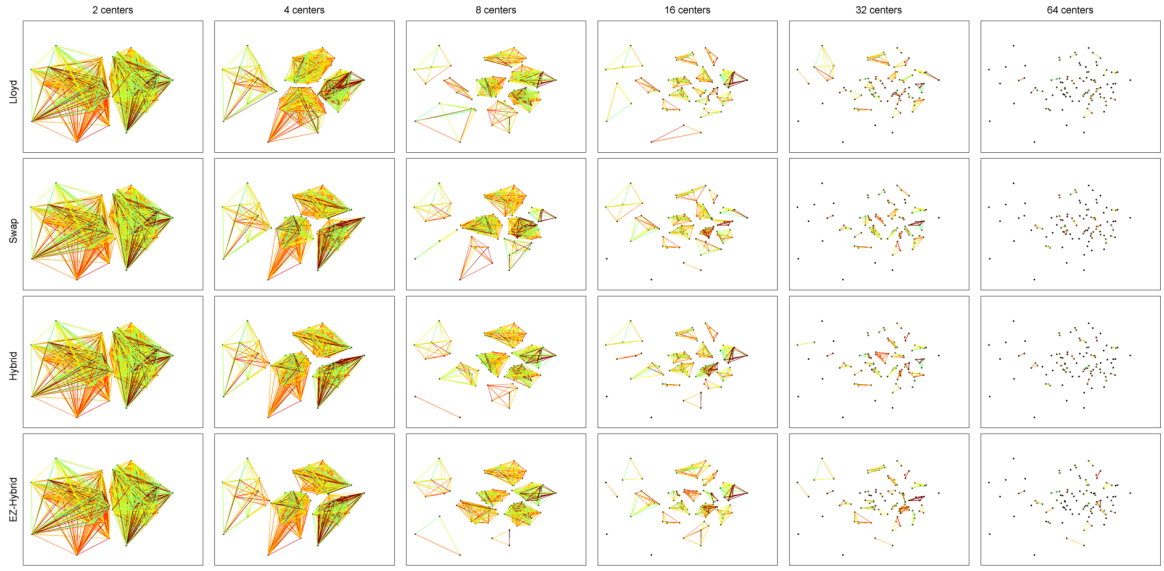


Figure D.15: Gauss node distribution using DYSE map #4.

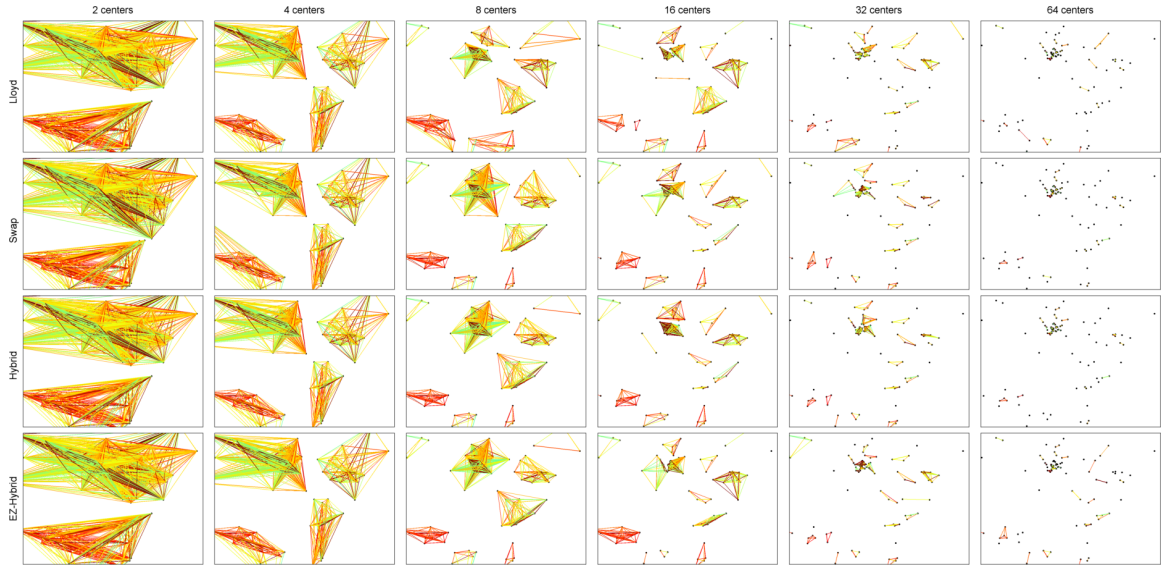


Figure D.16: Multi-cluster node distribution using DYSE map #4.

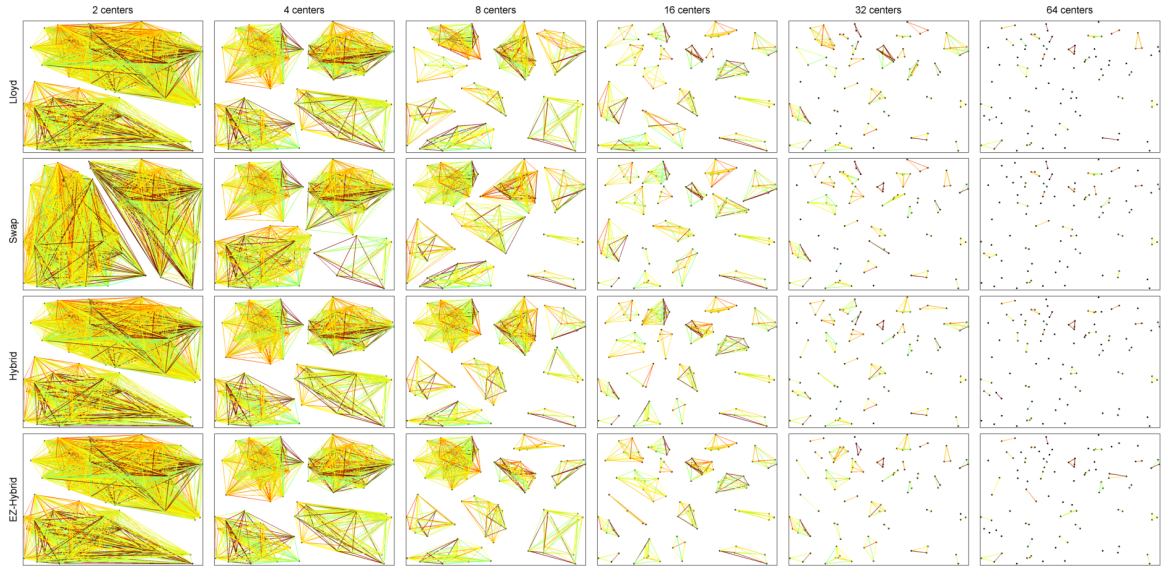


Figure D.17: Uniform node distribution using DYSE map #5.

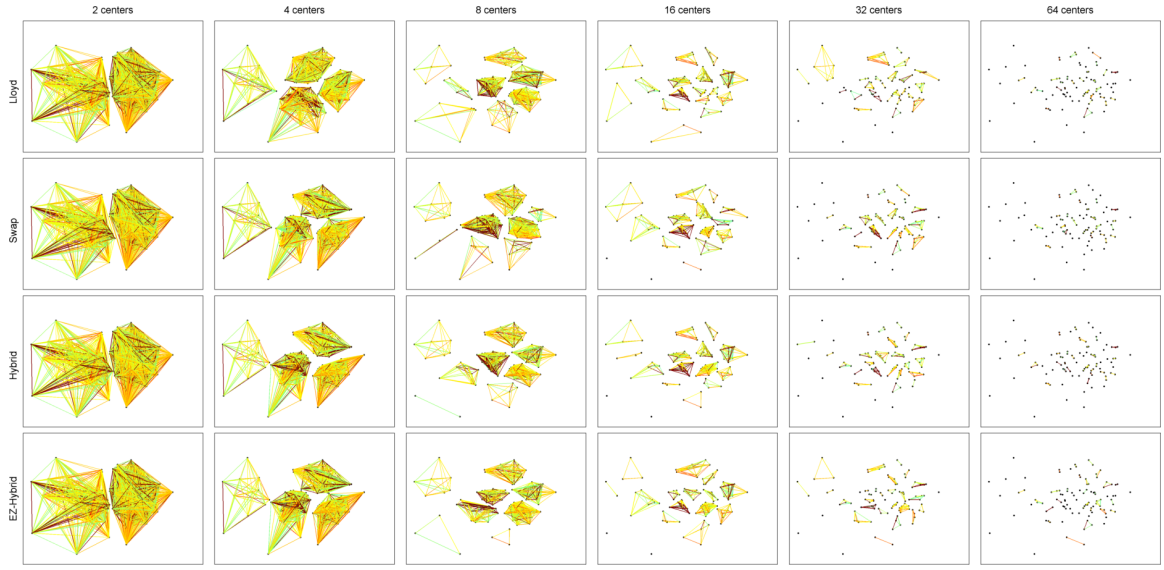


Figure D.18: Gauss node distribution using DYSE map #5.

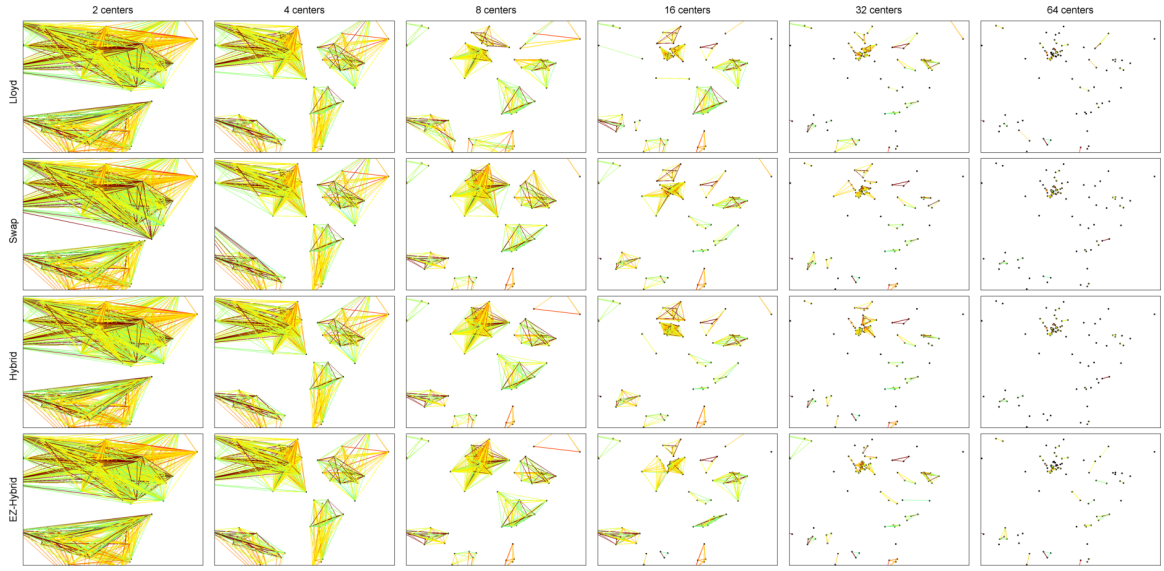


Figure D.19: Multi-cluster node distribution using DYSE map #5.

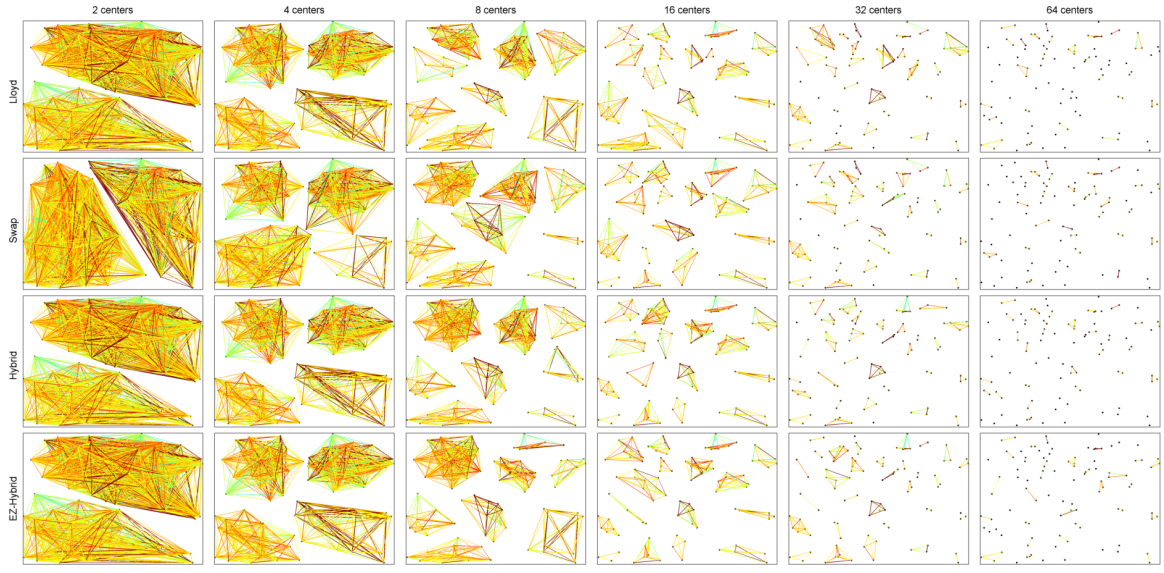


Figure D.20: Uniform node distribution using DYSE map #6.

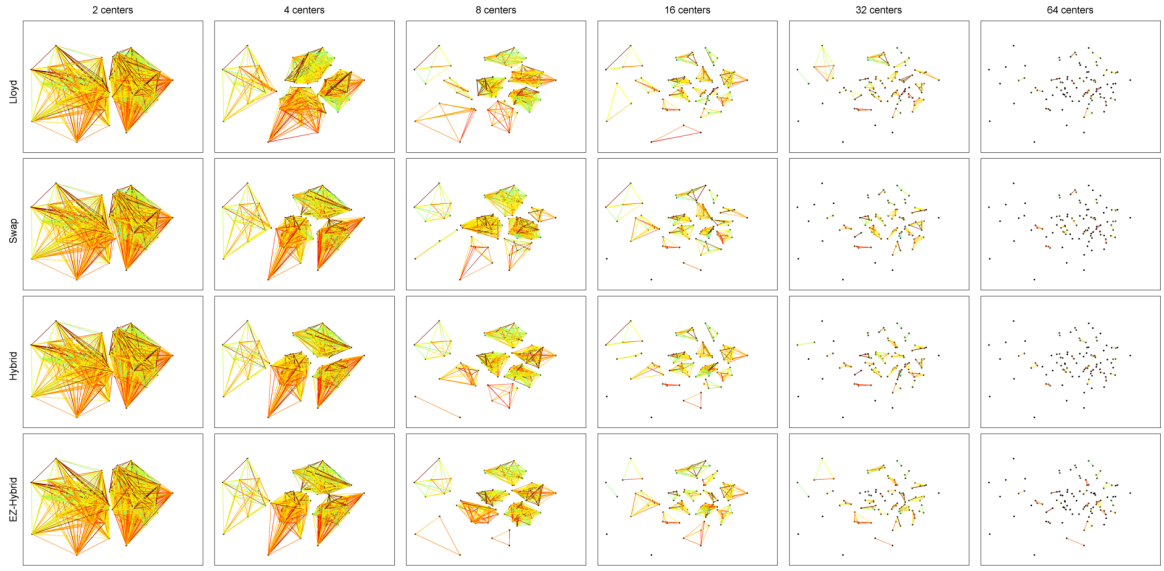


Figure D.21: Gauss node distribution using DYSE map #6.

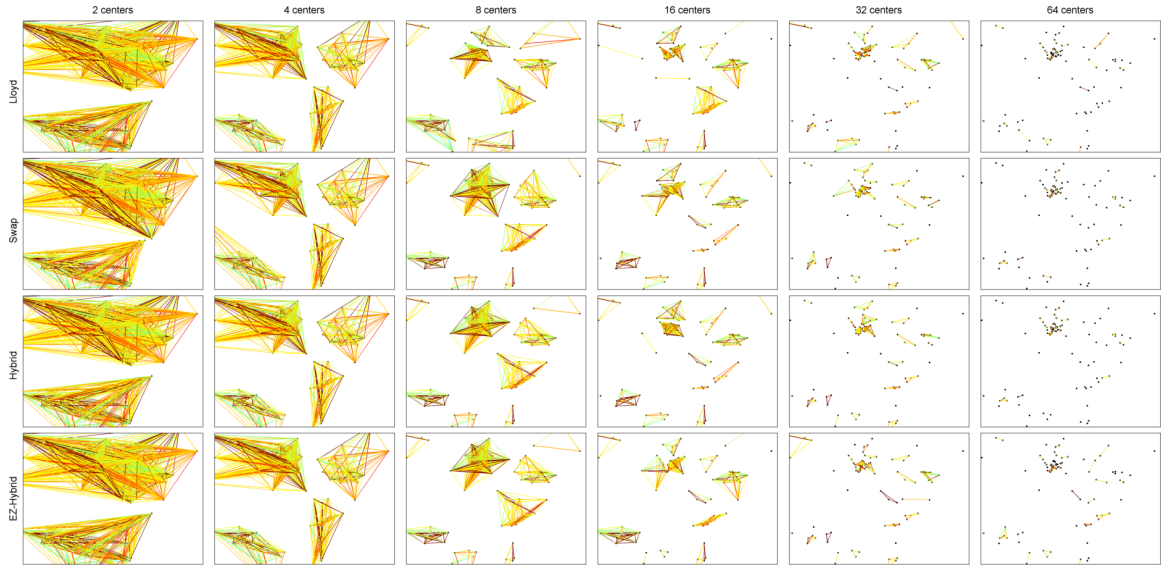


Figure D.22: Multi-cluster node distribution using DYSE map #6.

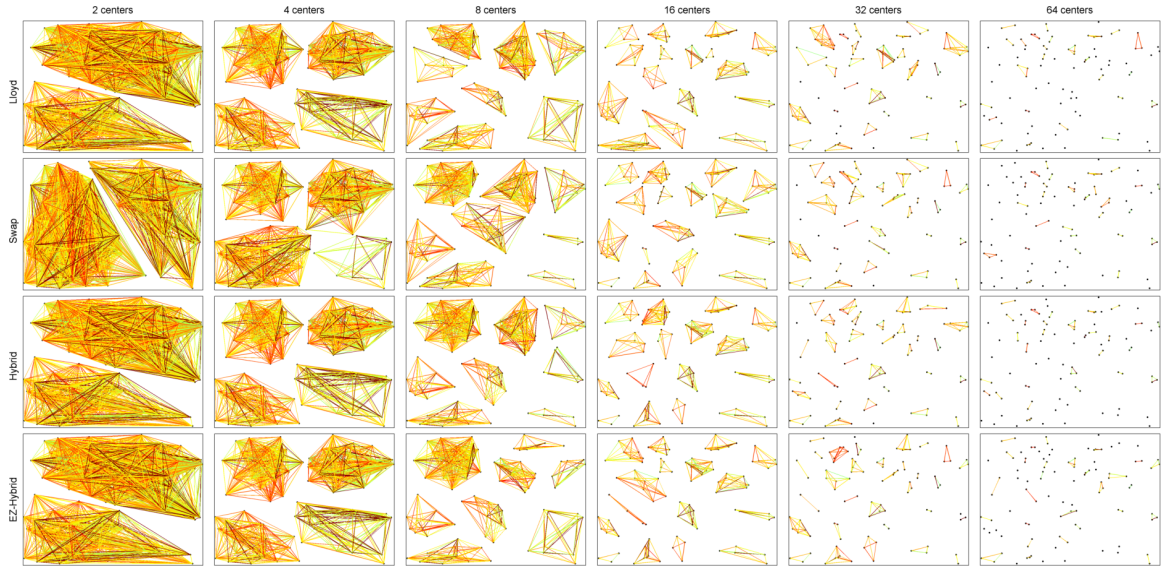


Figure D.23: Uniform node distribution using DYSE map #7.

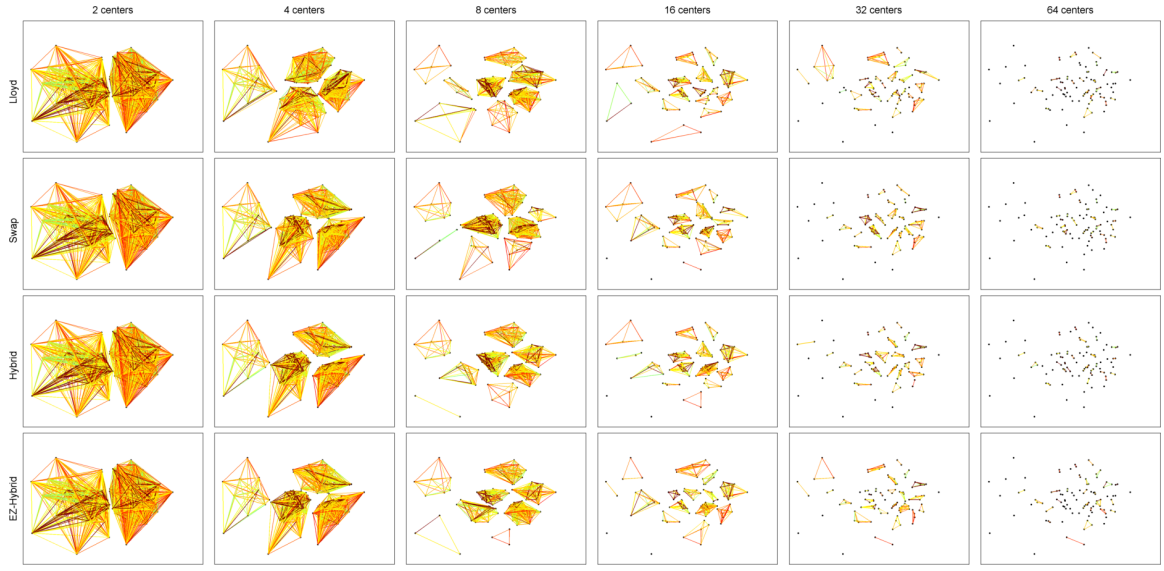


Figure D.24: Gauss node distribution using DYSE map #7.

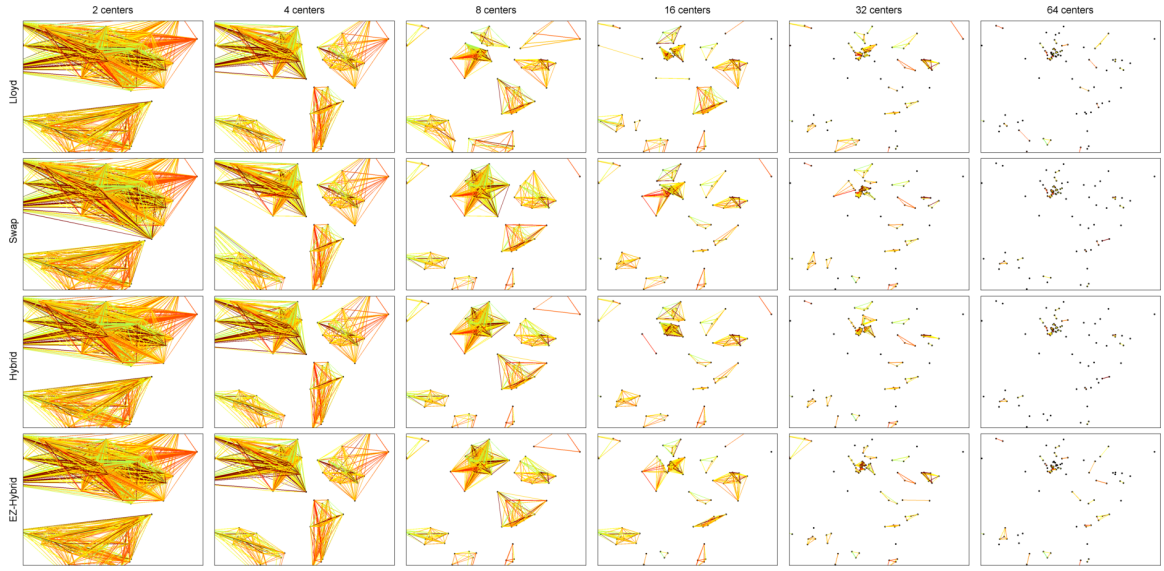


Figure D.25: Multi-cluster node distribution using DYSE map #7.

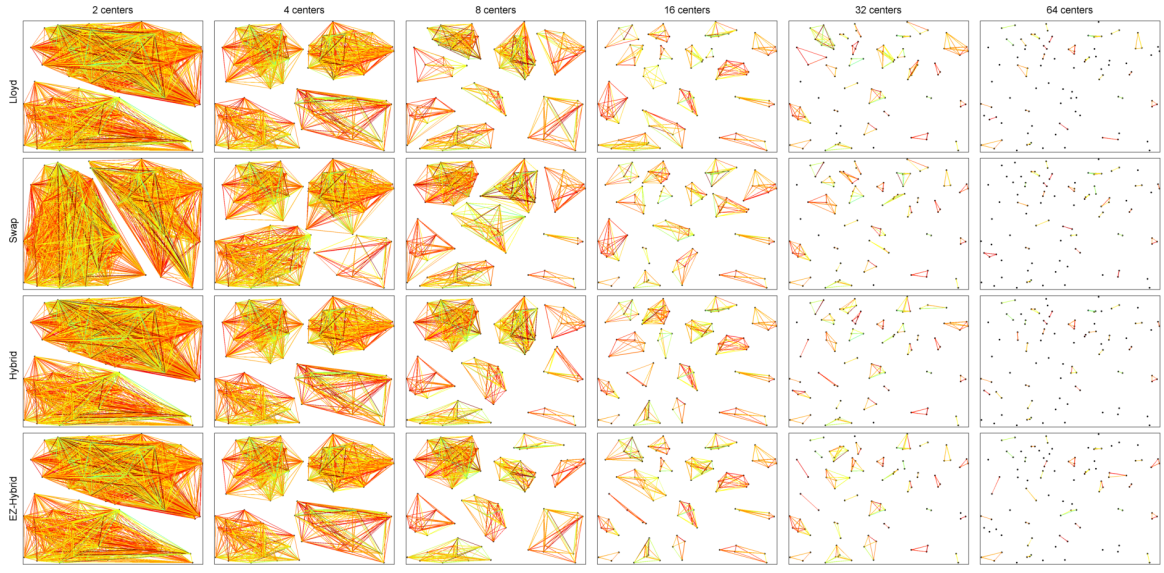


Figure D.26: Uniform node distribution using DYSE map #8.

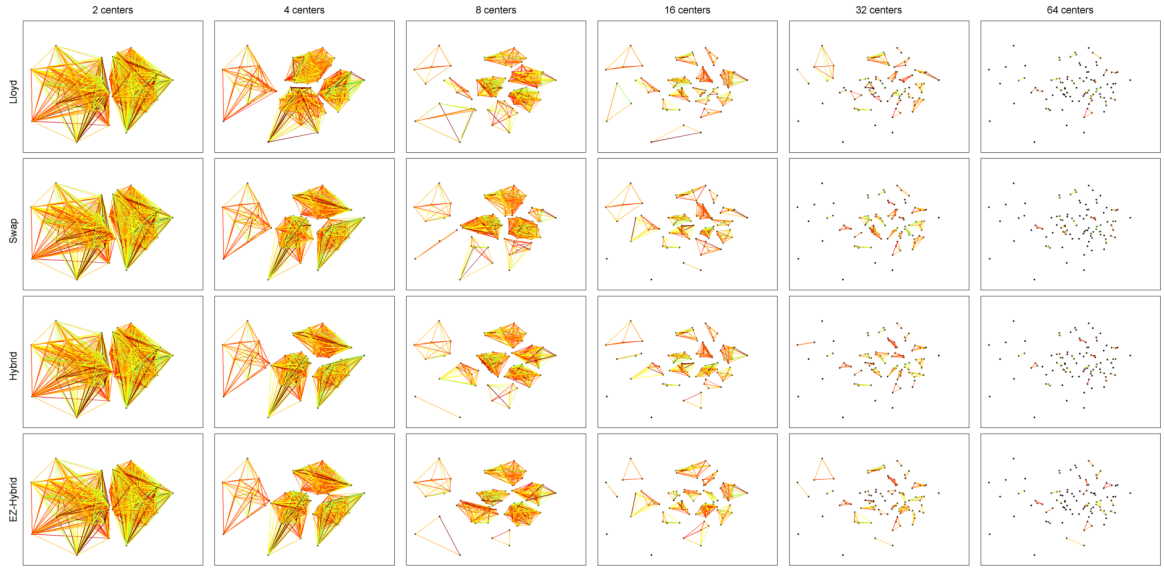


Figure D.27: Gauss node distribution using DYSE map #8.

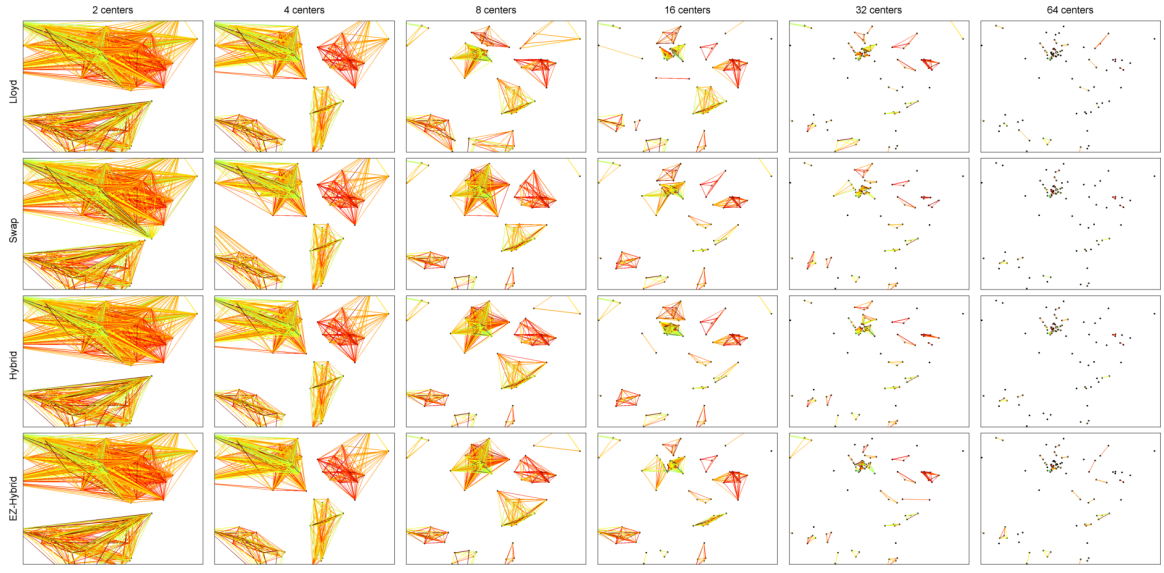


Figure D.28: Multi-cluster node distribution using DYSE map #8.

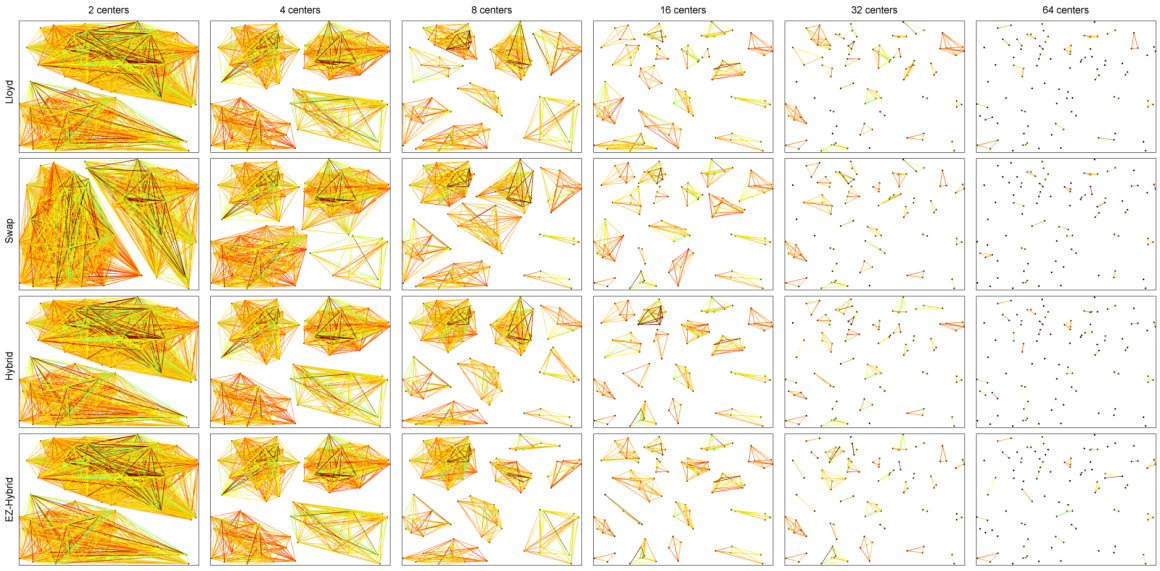


Figure D.29: Uniform node distribution using DYSE map #9.

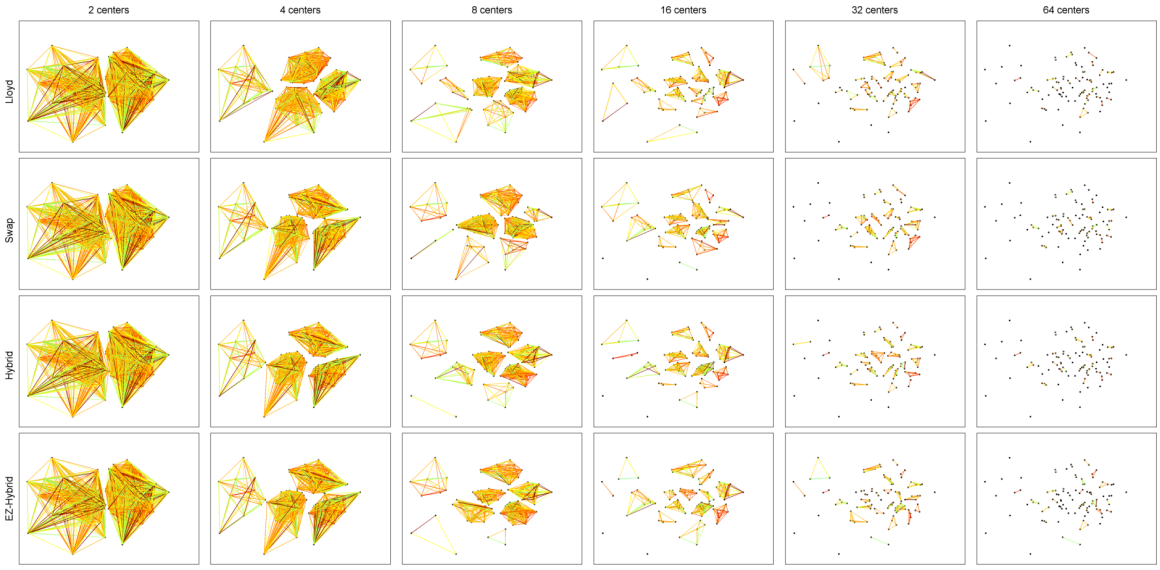


Figure D.30: Gauss node distribution using DYSE map #9.

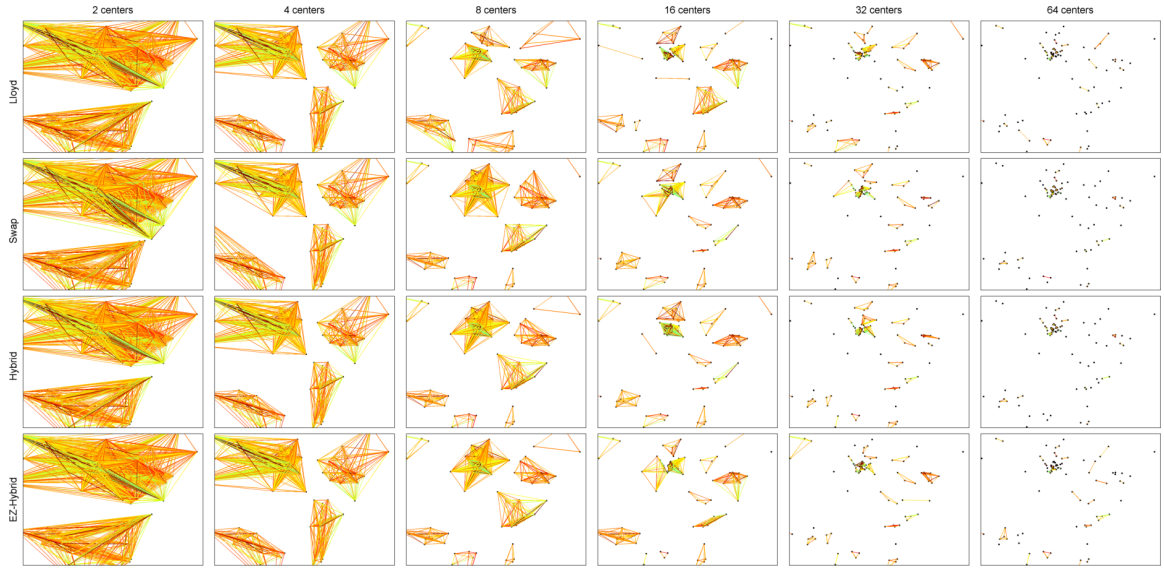


Figure D.31: Multi-cluster node distribution using DYSE map #9.

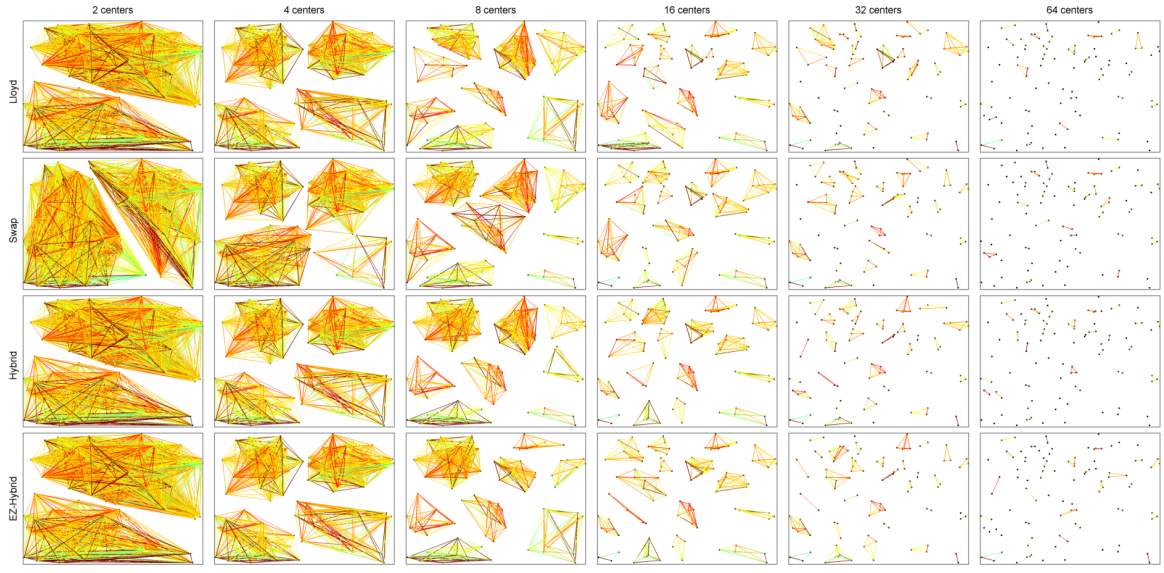


Figure D.32: Uniform node distribution using DYSE map #10.

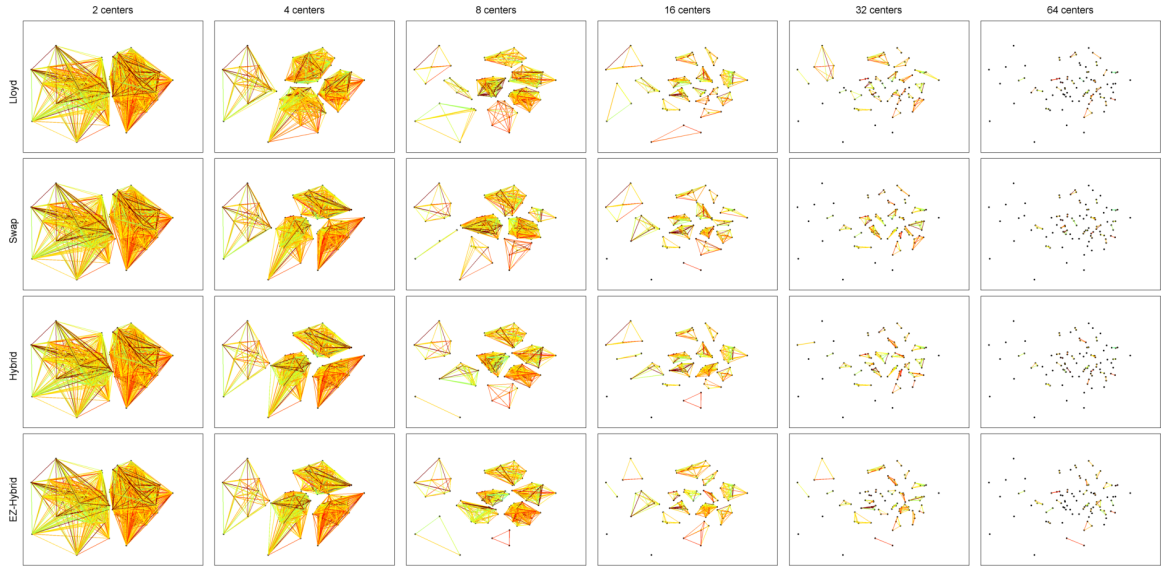


Figure D.33: Gauss node distribution using DYSE map #10.

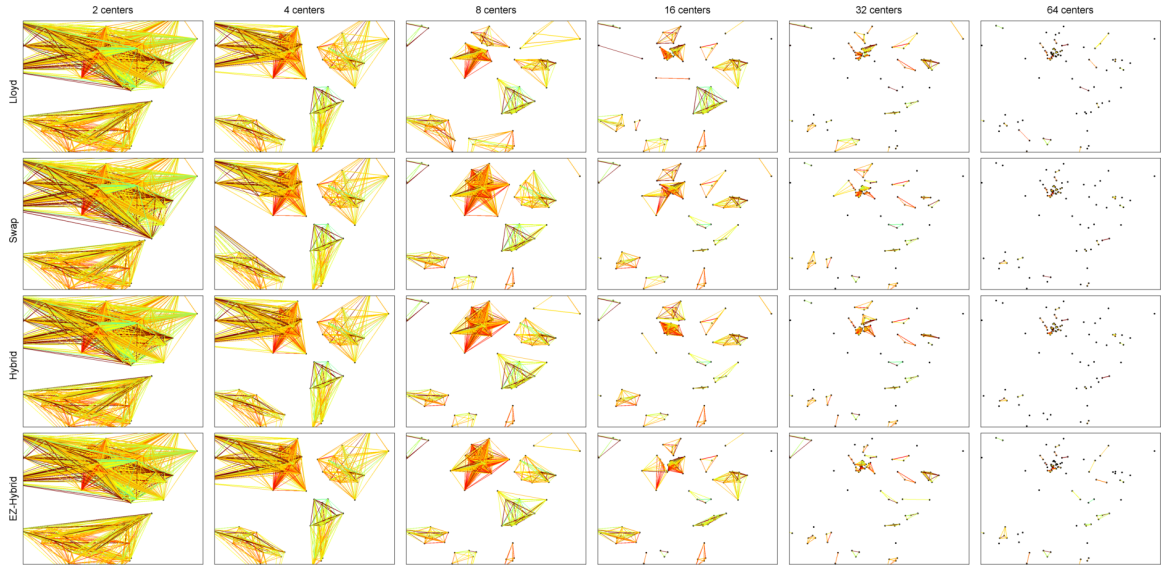


Figure D.34: Multi-cluster node distribution using DYSE map #10.

Appendix E: Additional ICSS Plots

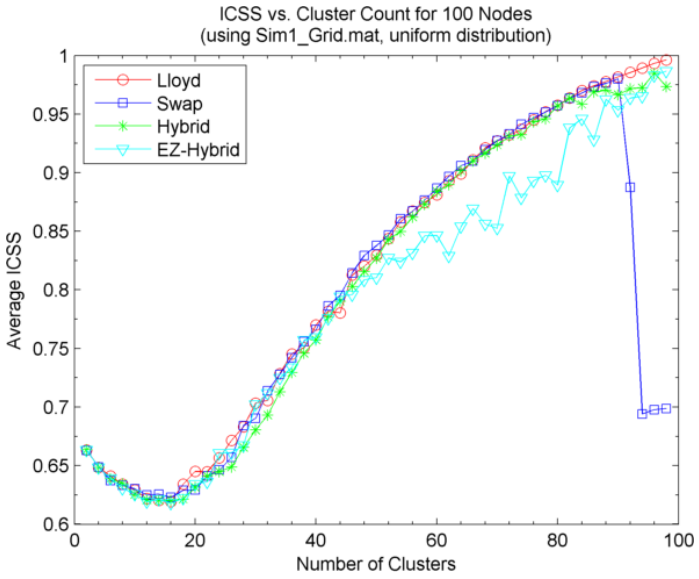


Figure E.1: ICSS for uniform distributions using DYSE map #1.

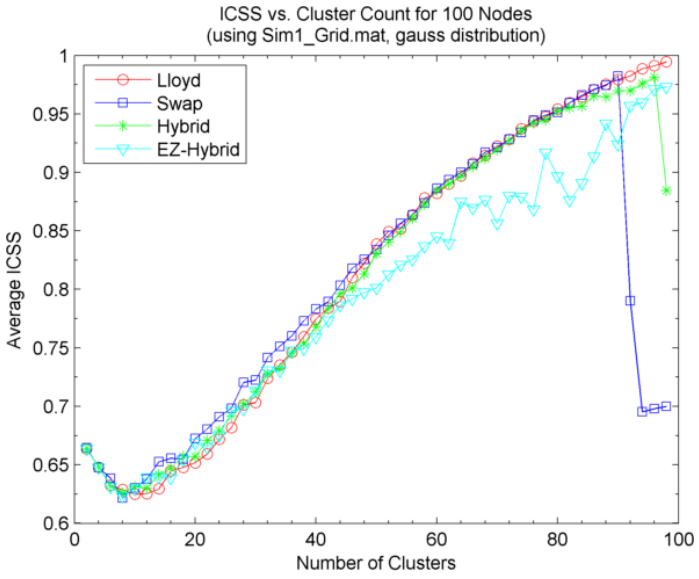


Figure E.2: ICSS for Gauss distributions using DYSE map #1.

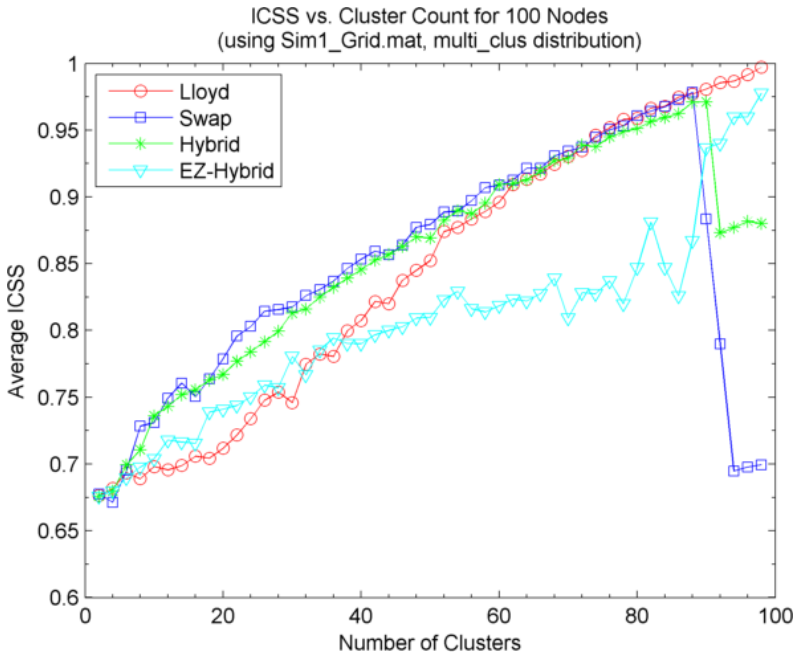


Figure E.3: ICSS for multi-cluster distributions using DYSE map #1.

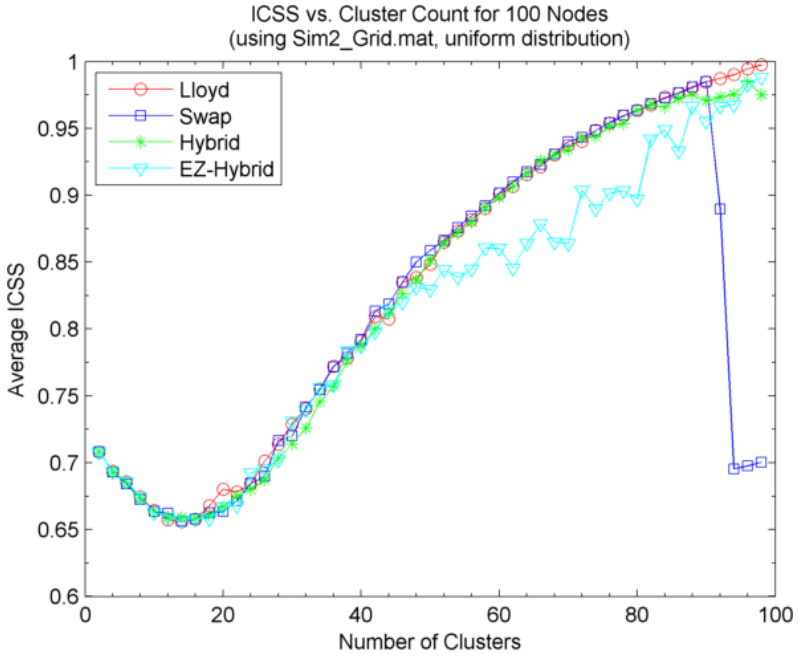


Figure E.4: ICSS for uniform distributions using DYSE map #2.

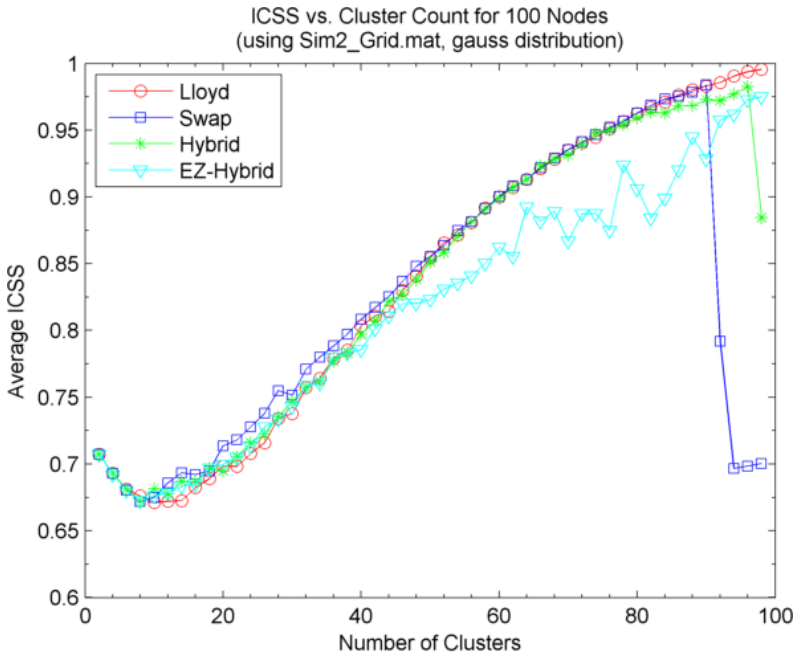


Figure E.5: ICSS for Gauss distributions using DYSE map #2.

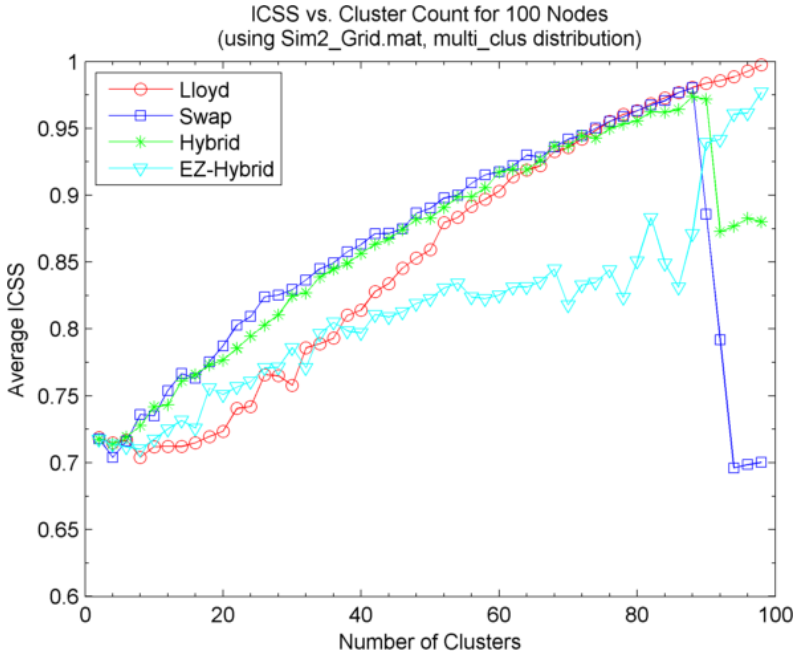


Figure E.6: ICSS for multi-cluster distributions using DYSE map #2.

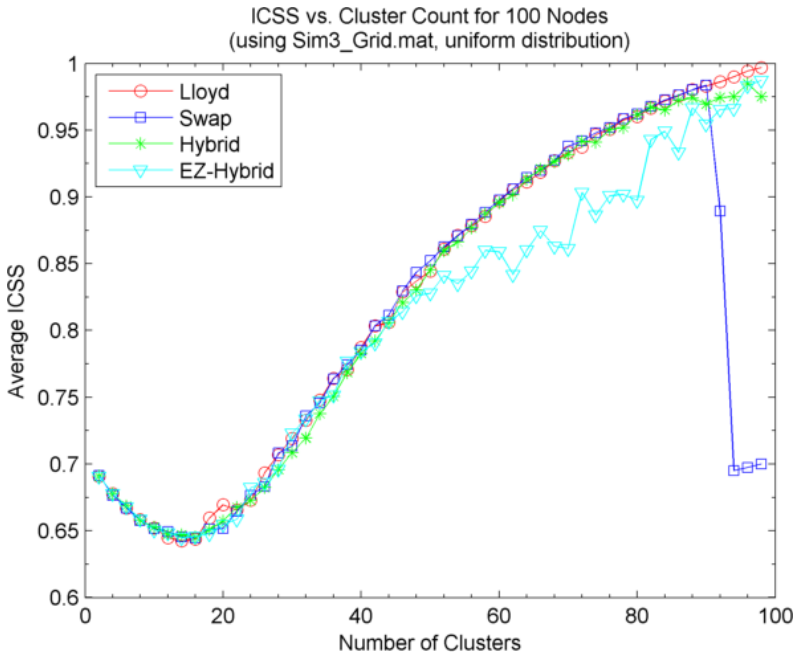


Figure E.7: ICSS for uniform distributions using DYSE map #3.

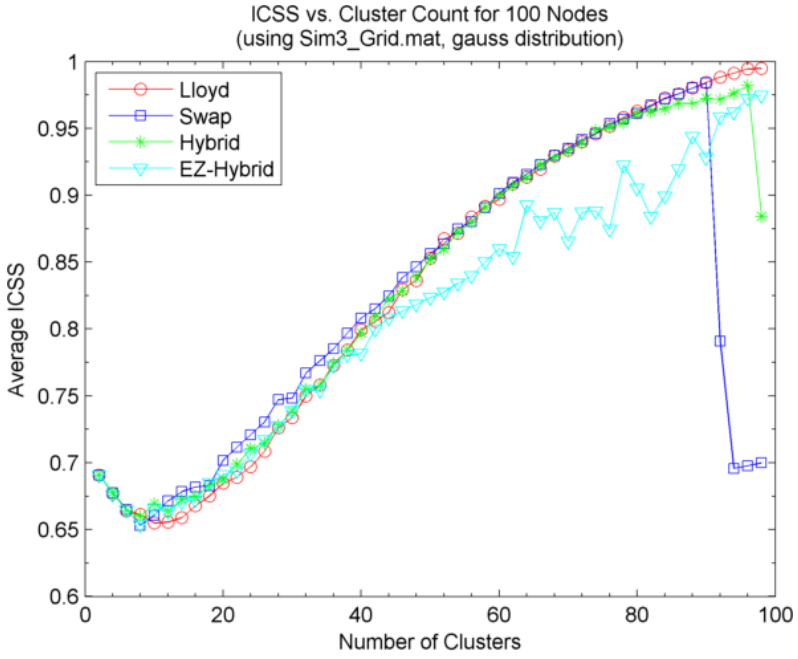


Figure E.8: ICSS for Gauss distributions using DYSE map #3.

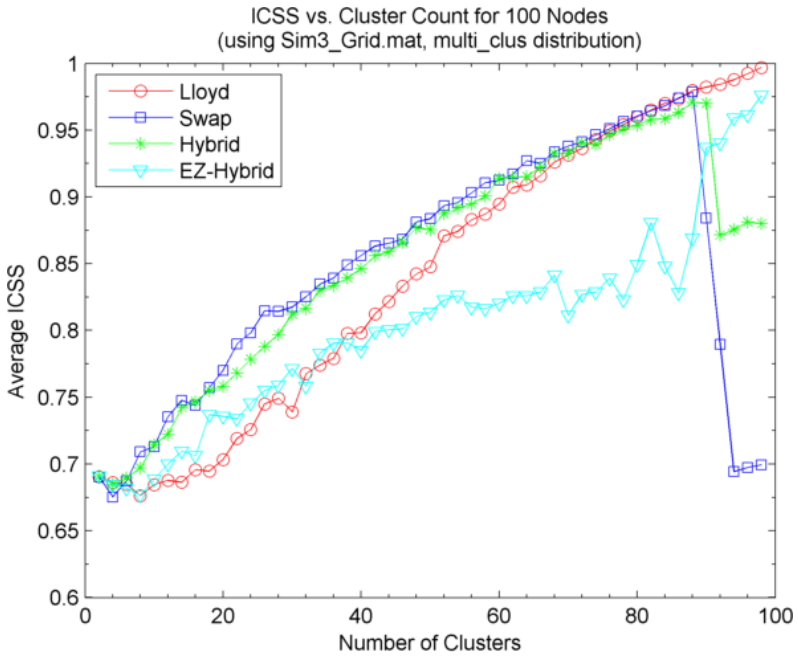


Figure E.9: ICSS for multi-cluster distributions using DYSE map #3.

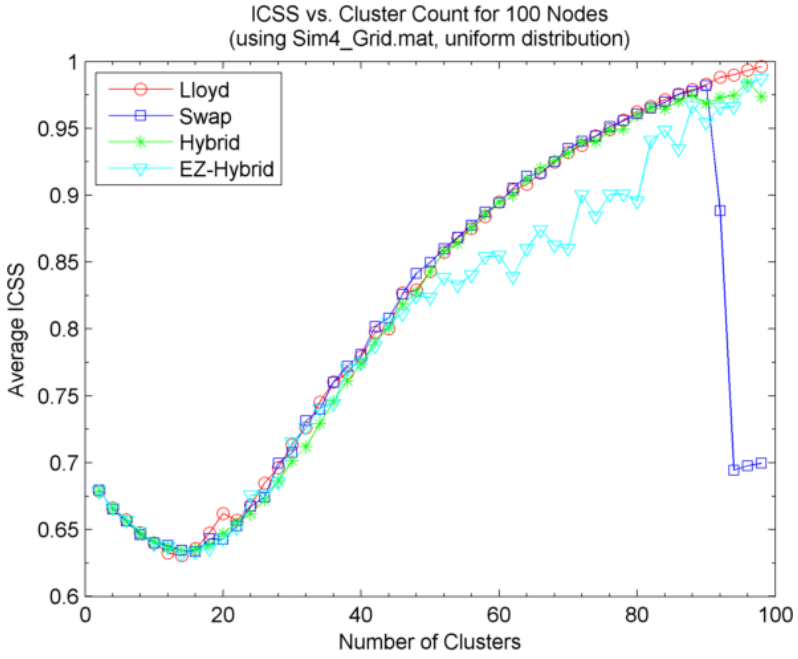


Figure E.10: ICSS for uniform distributions using DYSE map #4.

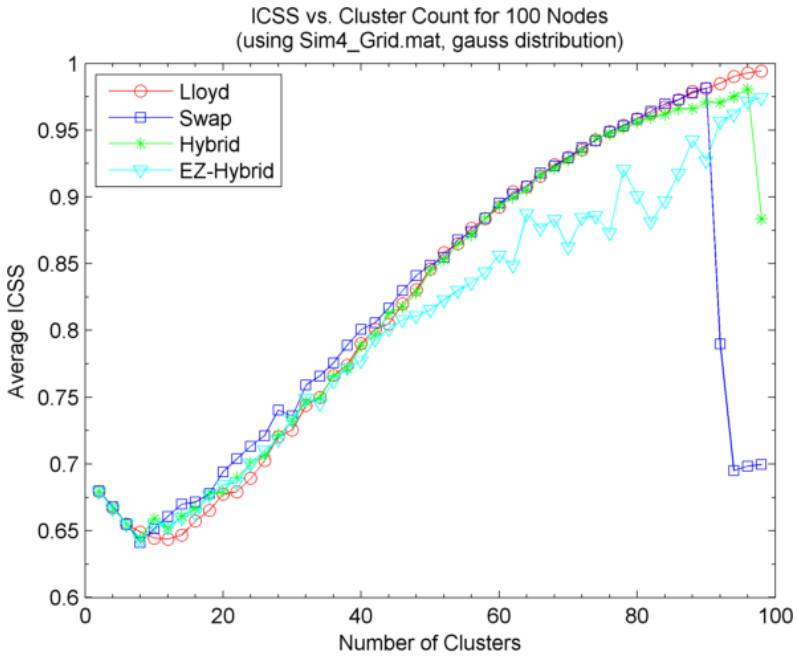


Figure E.11: ICSS for Gauss distributions using DYSE map #4.

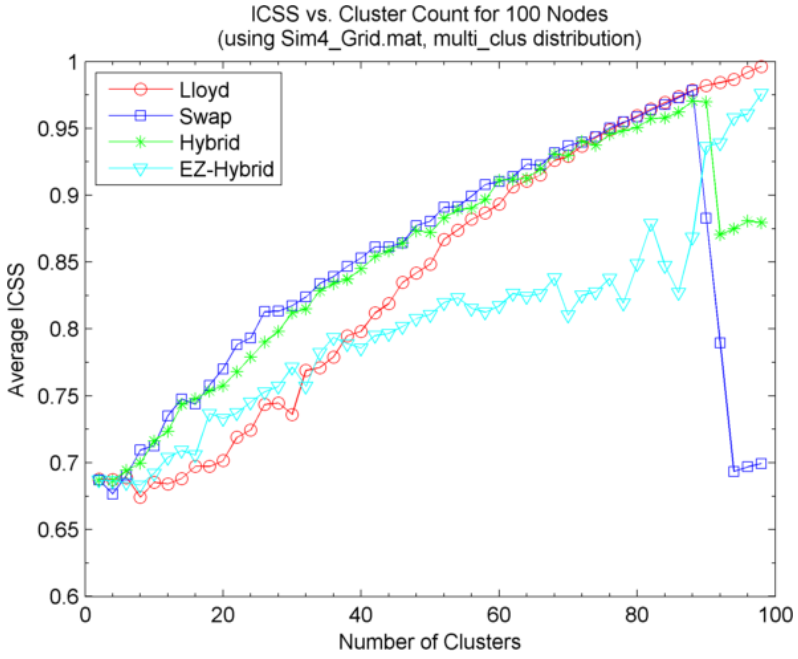


Figure E.12: ICSS for multi-cluster distributions using DYSE map #4.

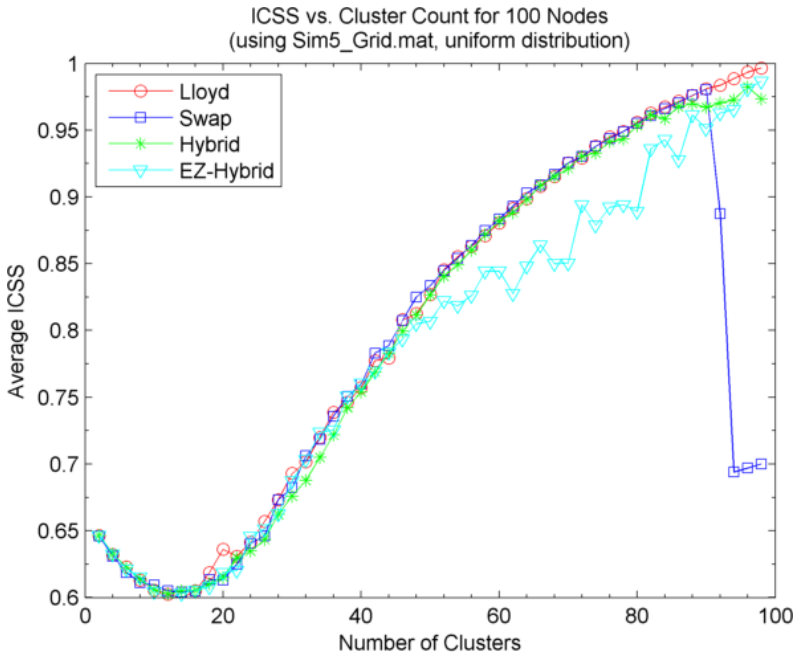


Figure E.13: ICSS for uniform distributions using DYSE map #5.

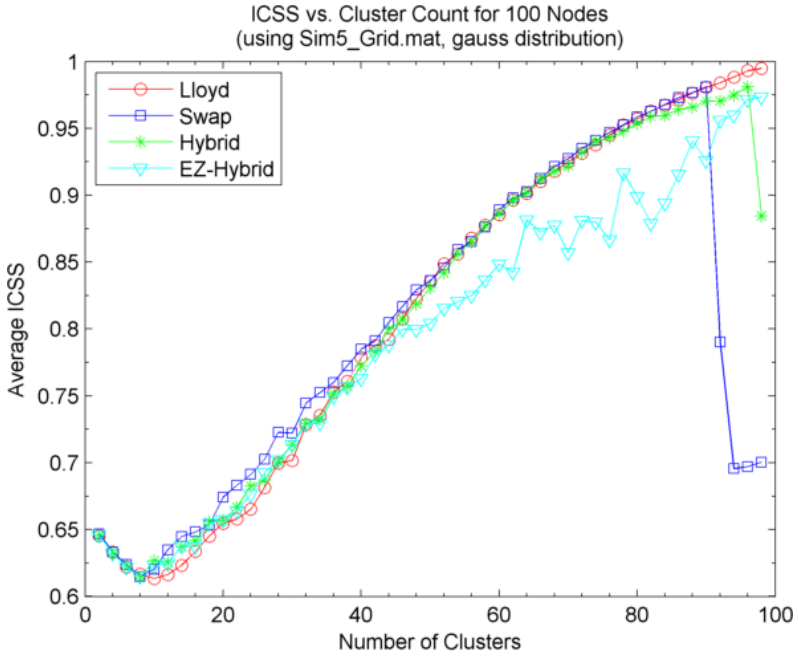


Figure E.14: ICSS for Gauss distributions using DYSE map #5.

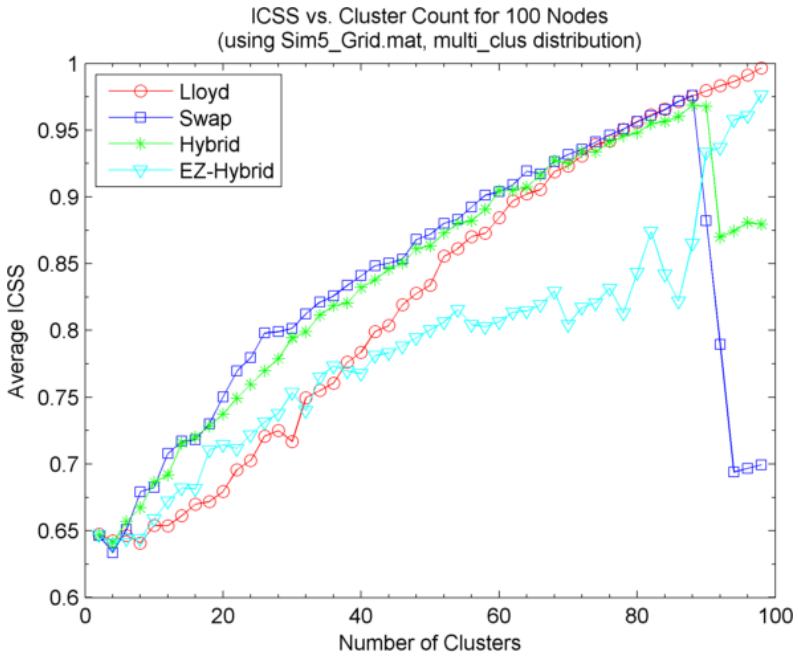


Figure E.15: ICSS for multi-cluster distributions using DYSE map #5.

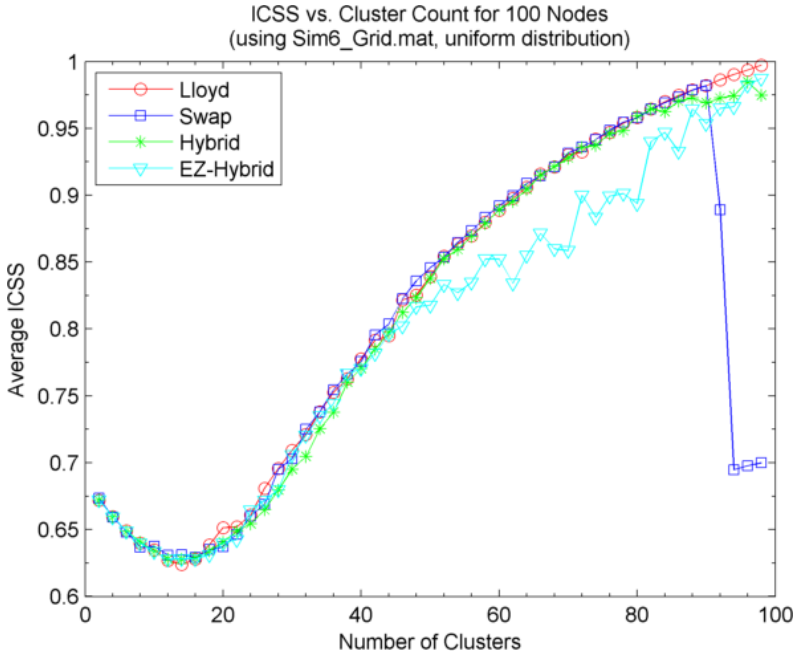


Figure E.16: ICSS for uniform distributions using DYSE map #6.

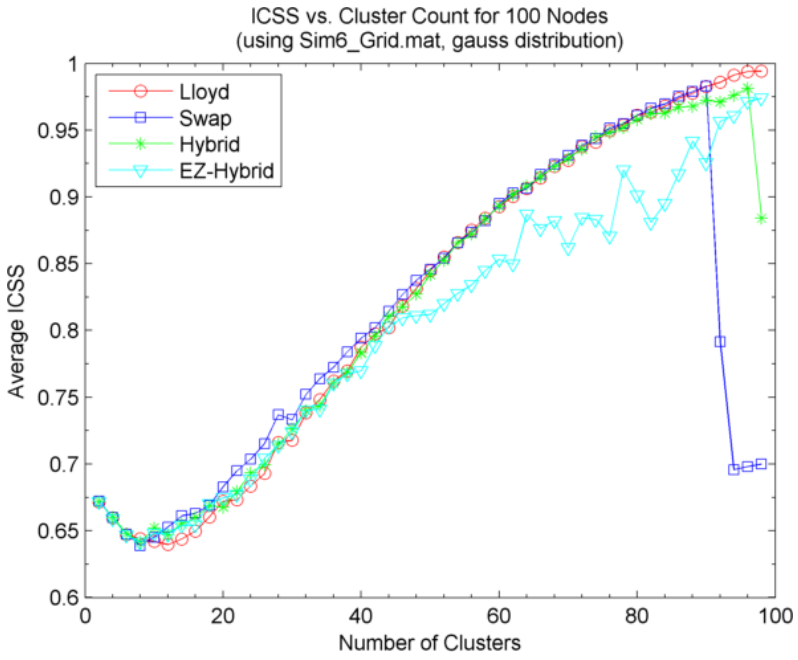


Figure E.17: ICSS for Gauss distributions using DYSE map #6.

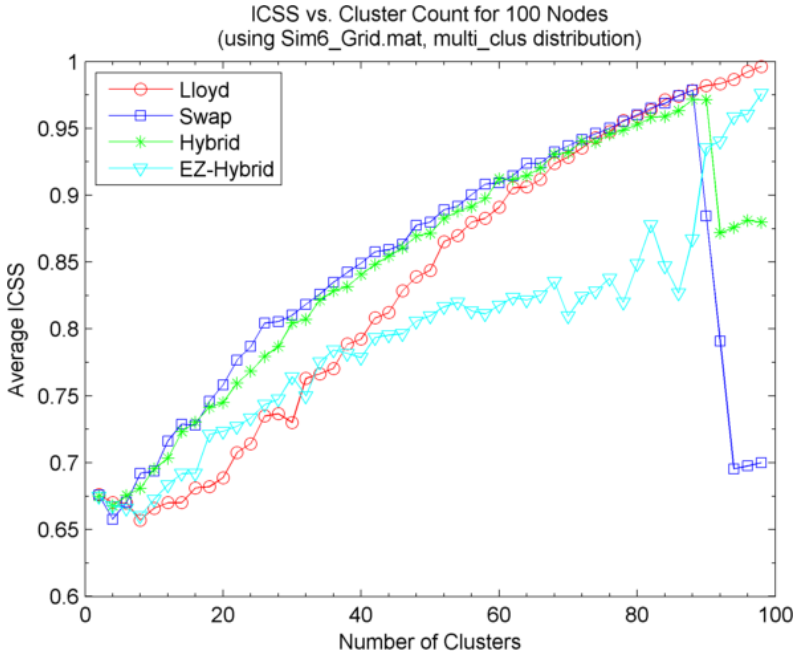


Figure E.18: ICSS for multi-cluster distributions using DYSE map #6.

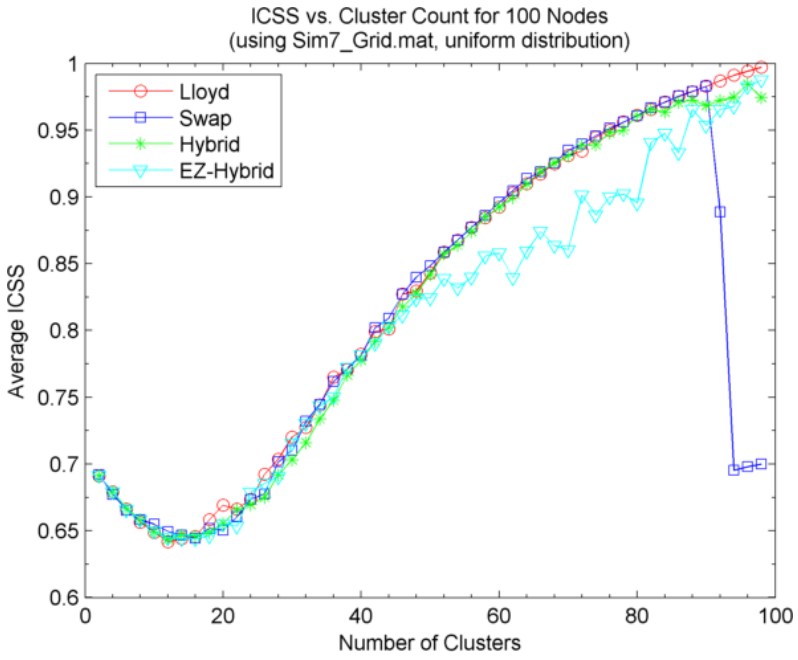


Figure E.19: ICSS for uniform distributions using DYSE map #7.

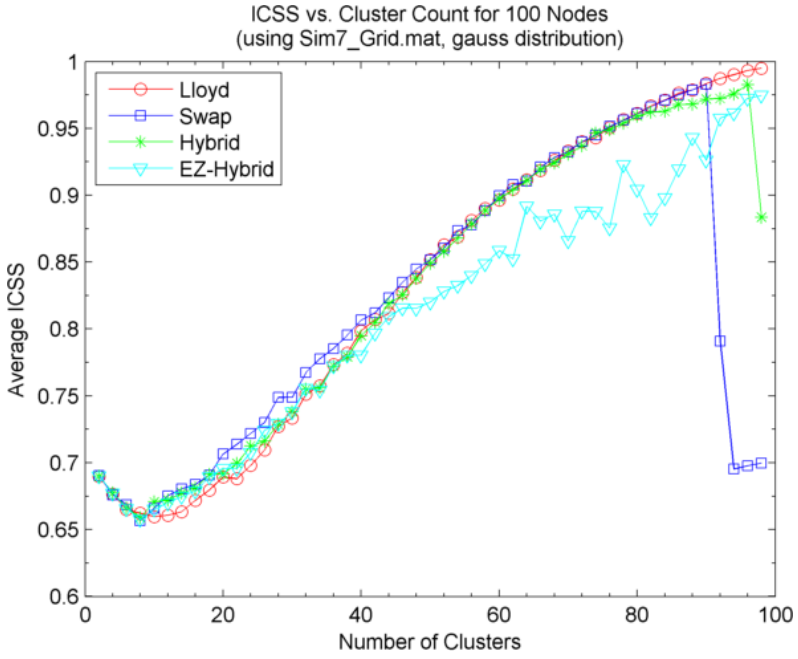


Figure E.20: ICSS for Gauss distributions using DYSE map #7.

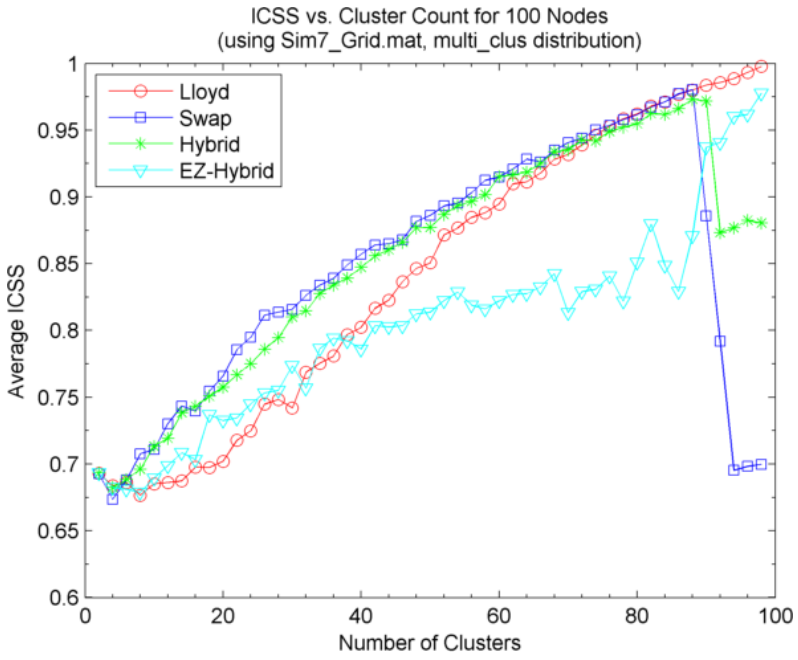


Figure E.21: ICSS for multi-cluster distributions using DYSE map #7.

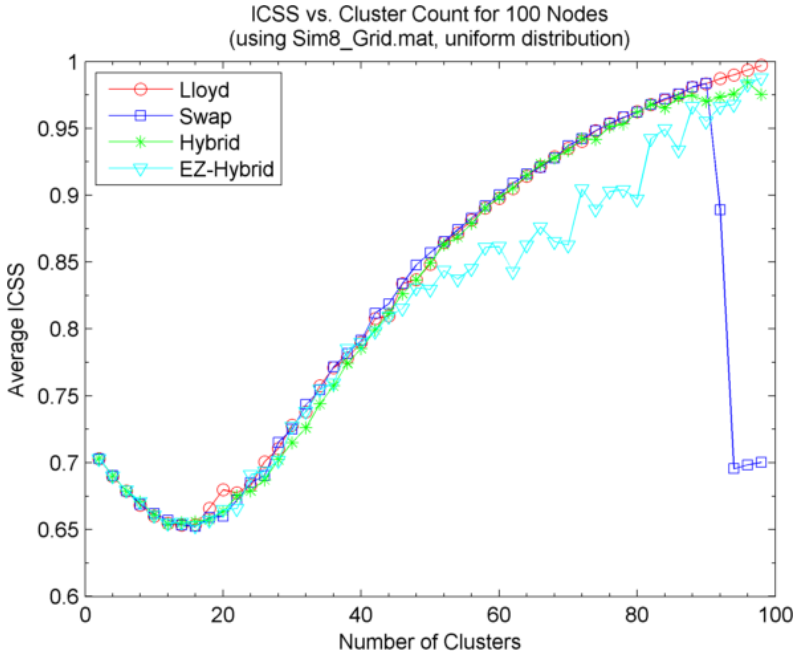


Figure E.22: ICSS for uniform distributions using DYSE map #8.

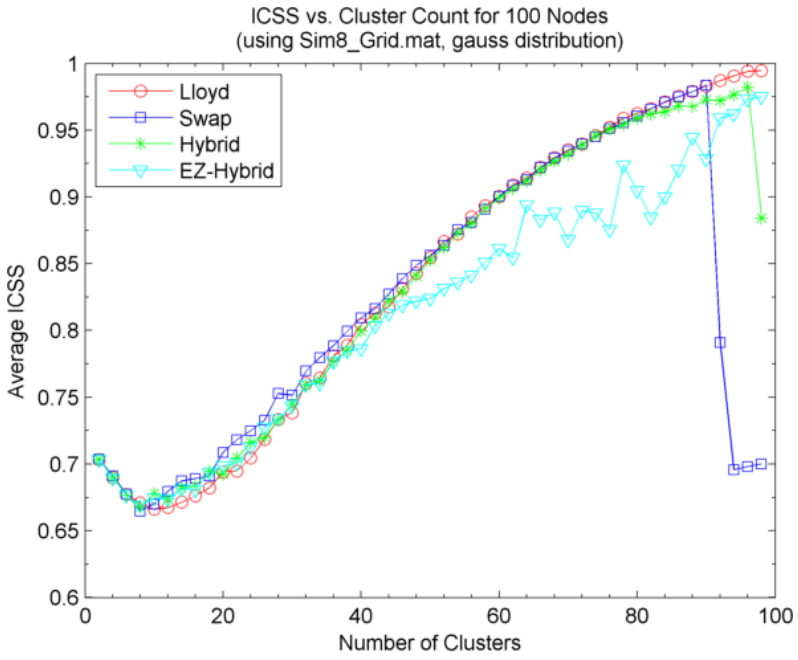


Figure E.23: ICSS for Gauss distributions using DYSE map #8.

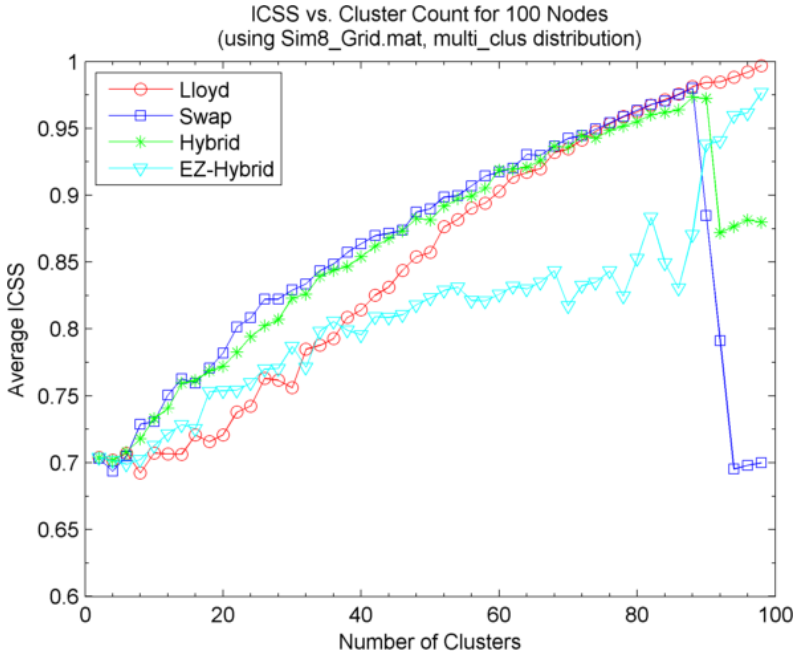


Figure E.24: ICSS for multi-cluster distributions using DYSE map #8.

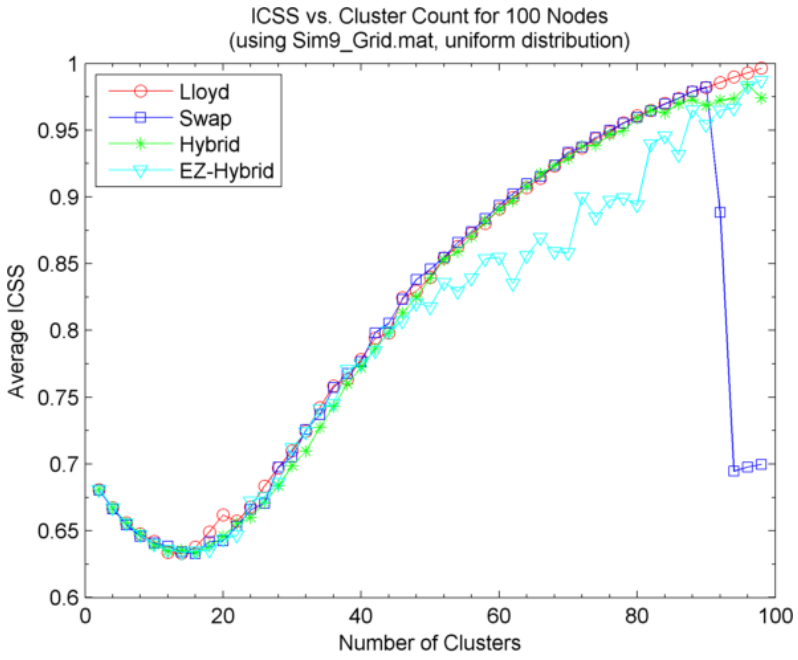


Figure E.25: ICSS for uniform distributions using DYSE map #9.

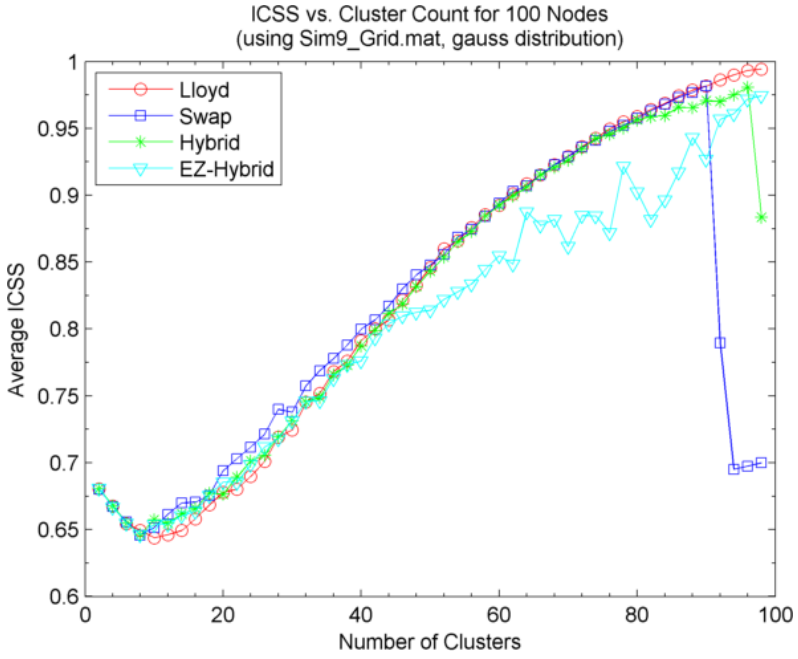


Figure E.26: ICSS for Gauss distributions using DYSE map #9.

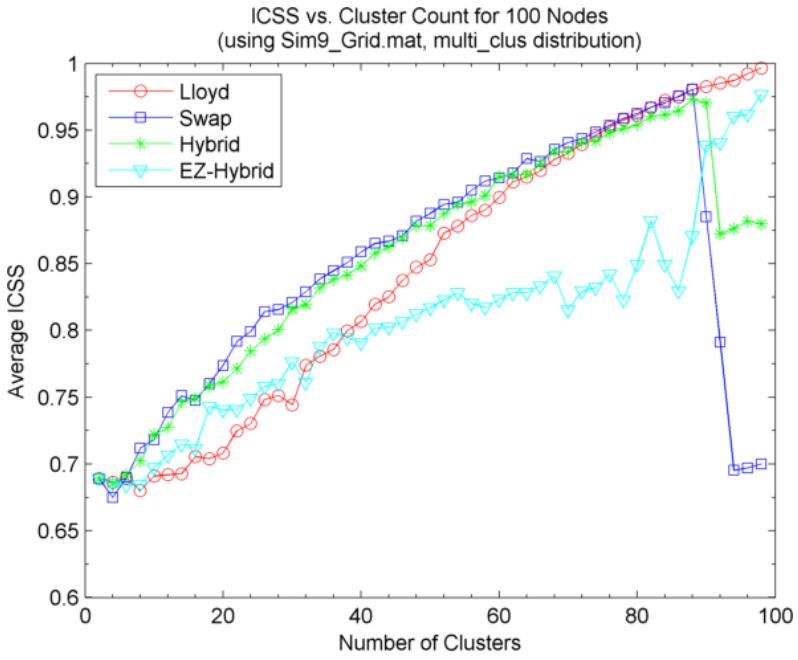


Figure E.27: ICSS for multi-cluster distributions using DYSE map #9.

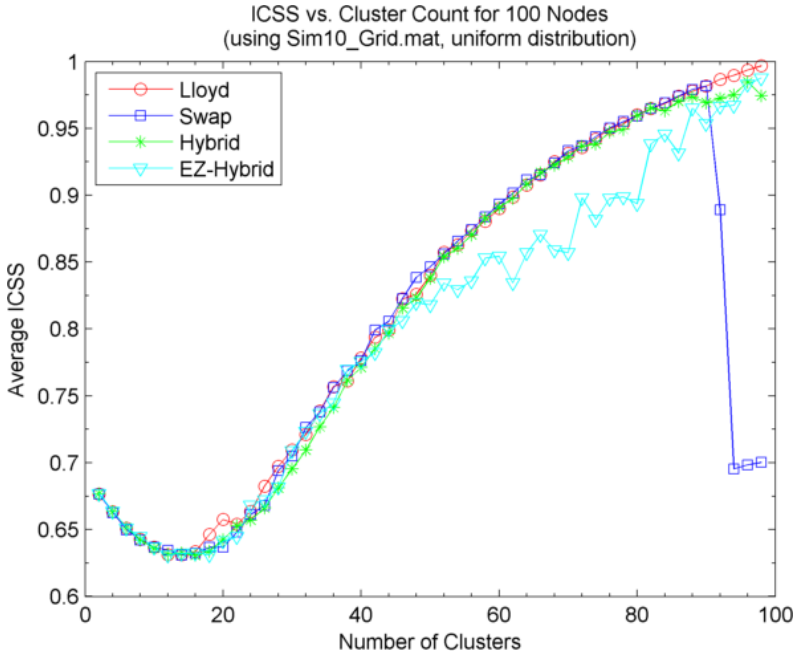


Figure E.28: ICSS for uniform distributions using DYSE map #10.

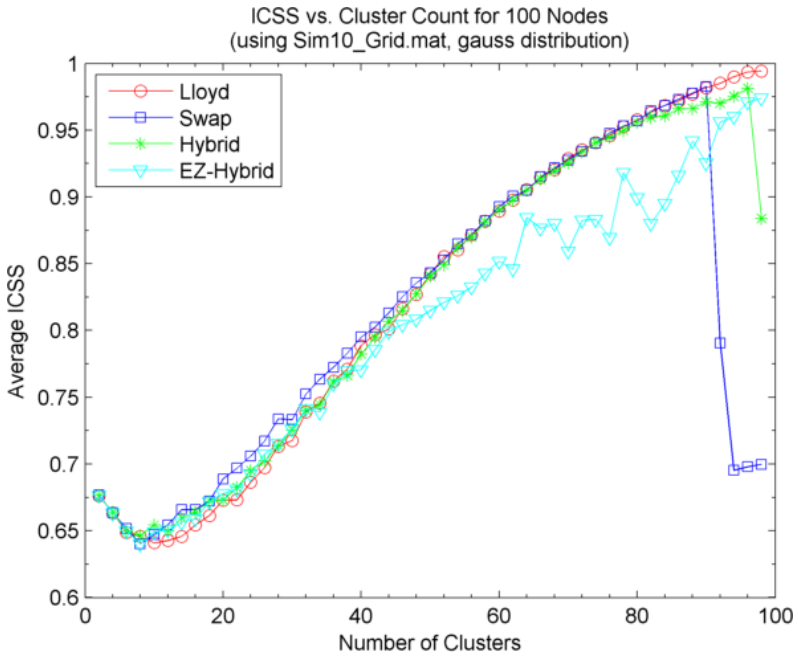


Figure E.29: ICSS for Gauss distributions using DYSE map #10.

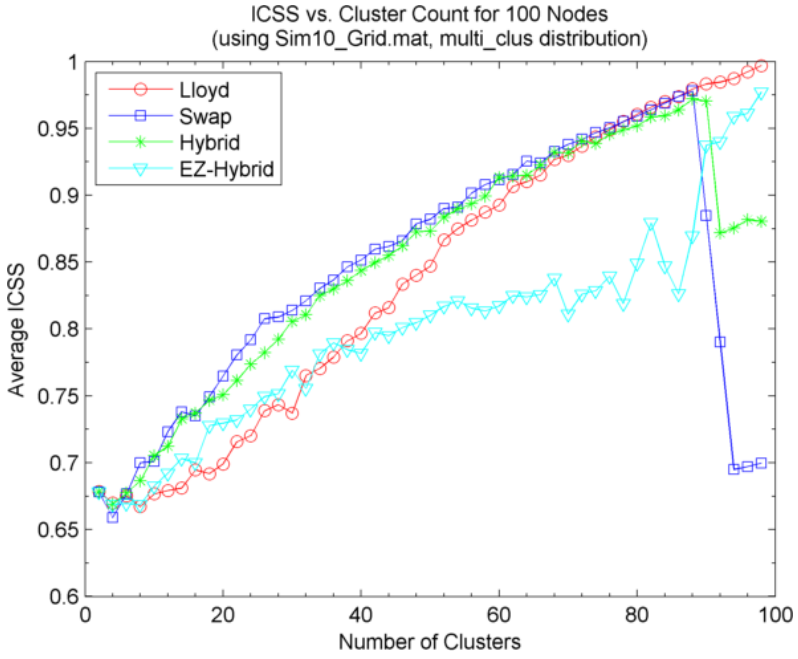


Figure E.30: ICSS for multi-cluster distributions using DYSE map #10.

Appendix F: MATLAB Code

MATLAB code was used to test the *k*-means clustering algorithm and its heuristics against various parameter configurations. In addition to the following files, `parseArgs.m` and `subaxis.m` (both available online) are also used. To load the spectrum, run `load_spectrum.m`. To run the test itself, run `kmeans_experiment.m`. For reference, the total MATLAB code base developed for this experiment is 863 lines, including vertical whitespace.

```
function spectrum = build_baseline_spectra(type, dim, bins)
    spectrum = zeros(dim, dim, bins);

    if strcmp(type, 'empty') == 1
        for i = 1:dim
            for j = 1:dim
                spectrum(i, j, :) = ones(bins, 1);
            end
        end
    elseif strcmp(type, 'disjoint') == 1
        interval = bins / (dim*dim)
        counter = 1;

        if interval < 1
            fprintf('Cannot build disjoint baseline--not enough ...
               bins.\n\n');
            spectrum = [];
            return;
        end

        for i = 1:dim
            for j = 1:dim
                temp = zeros(bins, 1);
                temp(counter*interval) = 1;
                spectrum(i, j, :) = temp;
                counter = counter + 1;
            end
        end
    elseif strcmp(type, 'full') == 1
        for i = 1:dim
            for j = 1:dim
                spectrum(i, j, :) = zeros(bins, 1);
            end
        end
    end
```

```
end
```

```
function clean_data()
    dos('del /Q .\commands\commands*');
    dos('del /Q .\output\output*');
    dos('del /Q .\input\input*');
end
```

```
function spectrum = decode_spectrum(mag, n_devices, info_grid, ...
    bins, dim)
    spectrum = zeros(dim, dim, bins);

    for x = 1:dim
        for y = 1:dim
            for d = 1:n_devices
                if info_grid(x, y) ~= 0
                    if mag(d, info_grid(x, y), :) ~= zeros(1, 1, bins)
                        spectrum(x, y, :) = mag(d, info_grid(x, ...
                            y), :);
                        break;
                    end
                end
            end
        end
    end
end
```

```
function intra_cluster_similarity = draw_clusters(map_number, ...
    distro, v_n_centers, map_assigned, n_nodes, rems, baseline, ...
    plot_clusters)
    heuristics = char('Lloyd', 'Swap', 'Hybrid', 'EZ-Hybrid');
    n_heuristics = 4;
    plot_number = 1;
    intra_cluster_similarity = zeros(numel(v_n_centers), ...
        n_heuristics);

    distros = {'Uniform' 'Gauss' 'Multi-Cluster'};%; 'co_gauss';
              %'co_laplace'; 'clus_gauss'; 'clus_orth_flats';
              %'clus_ellipsoids'; 'multi_clus'];

    for j = 1:n_heuristics
        fprintf('\n\tProcessing heuristic: %s\n', heuristics(j,:));
        fprintf('\t  Cluster counts drawn: ');

        for i = 1:numel(v_n_centers)
            cluster_similarities = zeros(v_n_centers(i), 2);
            cluster_similarities(:, 2) = ones(v_n_centers(i), 1);

```

```
19              if plot_clusters == 1
20                  set(gca, 'Units', 'normal');
21                  subaxis(n_heuristics, numel(v_n_centers), ...
                        plot_number, ...
22                          'Spacing', 0.01, 'MT', 0.05, 'MR', 0.02, ...
                               'MB', 0.05, 'ML', 0.02);
23
24                  if j == 1
25                      title(sprintf('%d centers', v_n_centers(i)));
26                  end
27
28                  if i == 1
29                      ylabel(heuristics(j,:));
30                  end
31
32                  hold on
33
34                  %centers_x_coords = centers(i, j, ...
                        1:v_n_centers(i), 1);
35                  %centers_y_coords = centers(i, j, ...
                        1:v_n_centers(i), 2);
36                  %scatter(centers_x_coords(:), ...
                        centers_y_coords(:), '.r');
37
38                  nodes_x_coords = map_assigned(i, j, 1, :);
39                  nodes_y_coords = map_assigned(i, j, 2, :);
40                  scatter(nodes_x_coords(:), nodes_y_coords(:), '.k');
41              end
42
43              for k = 1:n_nodes
44                  current_center = map_assigned(i, j, 3, k);
45                  current_rem = rems(k, :);
46
47                  cluster_similarities(current_center+1, 1) = ...
48                              cluster_similarities(current_center+1, ...
                                    1) + 1;
49
50                  for m = k+1:n_nodes
51                      compare_center = map_assigned(i, j, 3, m);
52                      compare_rem = rems(m, :);
53
54                      if compare_center == current_center
55                          similarity = 1.0-pdist([current_rem; ...
                                compare_rem], 'Hamming');
56
57                          cluster_similarities(current_center+1, 2) ...
                                = ...
58                              cluster_similarities(current_center+1, ...
                                    2) + similarity;
59
60                          if plot_clusters == 1
61                              x1 = map_assigned(i, j, 1, k);
```

```
                                y1 = map_assigned(i, j, 2, k);
                                x2 = map_assigned(i, j, 1, m);
                                y2 = map_assigned(i, j, 2, m);

                                colors = colormap();
                                plot([x1 x2], [y1 y2], 'Color', ...
                                    colors(ceil((63*similarity)) + 1, :));
                            end
                        end
                    end
                end

                temp_similarity = cluster_similarities(:, 1);

                for k = 1:v_n_centers(i)
                    if cluster_similarities(k, 1) ~= 0
                        cluster_similarities(k, 1) = ...
                            ceil(cluster_similarities(k, 1)^2 / 2);
                        temp_similarity(k) = cluster_similarities(k, ...
                            2) / cluster_similarities(k, 1);
                    end
                end

                %dot(cluster_similarities(:, 1), ...
                    cluster_similarities(:, 2)) / ...
                    sum(cluster_similarities(:, 1));
                %intra_cluster_similarity(i, j) = ...
                    dot(cluster_similarities(:, 1), ...
                    1./cluster_similarities(:, 2));
                intra_cluster_similarity(i, j) = mean(temp_similarity);

                if plot_clusters == 1
                    hold off

                    set(gcf,'Color','white')
                    set(gcf, 'Position', [1 1 1920 978])
                    axis([0 10 0 10]);
                    set(gca, 'xtick', []);
                    set(gca, 'ytick', []);
                    %set(gcf, 'Units', 'normal');
                    %set(gca, 'Position', get(gca, 'OuterPosition'));

                    plot_number = plot_number + 1;
                end

                fprintf('%d ', v_n_centers(i));
            end
        end

        if baseline == 0
```

```
105         export_fig(gcf, ...
                sprintf('./figures/Clustering/clustering%d_%s.png', ...
                map_number, char(distros(distro))));
106     else
107         export_fig(gcf, ...
                sprintf('./figures/Clustering/baseline%d.png', baseline));
108     end
109
110     close(gcf);
111
112     fprintf('\n');
113 end
114
115 function e = n_edges(n_vertices)
116     e = (n_vertices^2 / 2) - (n_vertices / 2);
117 end
```

```
1  function draw_distribution(map_number, n_nodes, centers, distro, ...
       center_count, dim, baseline)
2      distros = {'Uniform' 'Gauss' 'Multi-Cluster'};%; 'co_gauss';
3                 %'co_laplace'; 'clus_gauss'; 'clus_orth_flats';
4                 %'clus_ellipsoids'; 'multi_clus'];
5
6      seed = 1;
7
8      for m = 1:map_number
9          if m == 3
10             seed = seed + 1;
11         end
12
13         for d = 1:distro
14             fig = figure(1000 + (m-1)*distro + d);
15             set(fig, 'Position', [100 100 500 500])
16             gen_kmeans_commands(m, n_nodes, centers, d, ...
                   center_count, seed, baseline);
17             run_kmeans(m, center_count);
18             points = read_kmeans_points(m, n_nodes, dim, baseline);
19
20             hold on
21             h = scatter(points(:, 1), points(:, 2), '.k');
22             hChildren = get(h, 'Children');
23             set(hChildren, 'Markersize', 15)
24
25         %     title(sprintf('%s Distribution\nseed = %d', ...
               char(distros(distro)), seed_number))
26         %     xlabel('X Location')
27         %     ylabel('Y Location')
28
29             set(gcf,'Units','normal')
30             set(gcf,'Color','white')
31             axis([0 10 0 10])
```

```
32
33              set(gca,'Position',[0.05 0.05 0.91 0.91])
34              set(gca,'Box','on')
35
36              % x-axis parameters
37              set(gca,'XTickMode', 'manual')
38              set(gca,'XMinorTick', 'on')
39              set(gca,'XTick',0:10)
40              set(gca,'XAxisLocation','Top')
41
42              % y-axis parameters
43              set(gca,'YTickMode', 'manual')
44              set(gca,'YMinorTick', 'on')
45              set(gca,'YTick',0:10)
46              set(gca,'YDir','reverse');
47
48              grid on
49              set(gca,'GridLineStyle','-');
50              %set(gca,'XColor',[0.5 0.5 0.5]);
51              %set(gca,'YColor',[0.5 0.5 0.5]);
52
53              export_fig(gcf, ...
                    sprintf('.\\figures\\Node_maps\\%s_%d.png',char(distros(d)), ...
                    m));
54
55              close(gcf);
56
57              hold off
58              fprintf('|');
59          end
60
61          seed = seed + 1;
62      end
63
64      fprintf('\n');
65  end
```

```
1   function gen_kmeans_commands(map_number, n_nodes, centers, ...
        distro_indices, center_count, seed_number, baseline)
2       heuristics = {'lloyd', 'swap', 'hybrid', 'EZ-hybrid'};
3
4       for h = 1:numel(heuristics)
5           show_assignments = 'show_assignments yes';
6           kcenters = 'kcenters';
7
8           % Point generation commands
9           print_points = 'print_points yes';
10          distribution = 'distribution';
11          data_size = 'data_size';
12          gen_data_pts = 'gen_data_pts';
13          read_data_pts = 'read_data_pts';
```

```
14          seed = 'seed';
15
16          % Point distribution types
17          distros = {'uniform' 'gauss' 'multi_clus'};%; 'co_gauss';
18                     %'co_laplace'; 'clus_gauss'; 'clus_orth_flats';
19                     %'clus_ellipsoids'; 'multi_clus'];
20
21          % Point distribution type parameters
22          std_dev = 'std_dev';           % for gauss, clustered; ...
               def = 1
23          std_dev_lo = 'std_dev_lo';     % for clus_ellipsoids; def ...
               = 1
24          std_dev_hi = 'std_dev_hi';     % for clus_ellipsoids; def ...
               = 1
25          corr_coef = 'corr_coef';       % for co_gauss, ...
               co_laplace; def = 0.05
26          colors = 'colors';             % for clustered; def = 5
27          max_clus_dim = 'max_clus_dim'; % for clus_orth_flats, ...
               clus_ellipsoids;
28                                         %   def = 1
29
30          % Runtime command/parameters
31          run_kmeans = 'run_kmeans';
32
33          % System commands
34          quit = 'quit';
35
36          % Open the file for writing
37          fid = fopen(sprintf('./commands/commands%d_%d_%s', ...
               map_number, center_count, char(heuristics(h))), 'w');
38
39          % Write commands to the command input file
40          fprintf(fid, '%s\n', show_assignments);
41          fprintf(fid, '%s\n', print_points);
42          fprintf(fid, '%s %d\n', seed, seed_number);
43          fprintf(fid, '%s %d\n', data_size, n_nodes);
44
45          for i = distro_indices
46              % Set gauss std_dev
47              if i == 2 || i == 3
48                  fprintf(fid, '%s %f\n', std_dev, 0.3);
49              end
50
51              if i == 3
52                  fprintf(fid, '%s %d\n', colors, 2);
53              end
54
55              fprintf(fid, '%s\n', sprintf('%s %s', distribution, ...
                   char(distros(i))));
56
57              if baseline == 0
58                  fprintf(fid, '%s\n', gen_data_pts);
```

```
59              else
60                  fprintf(fid, '%s ./input/baseline%d', ...
                        read_data_pts, baseline);
61              end
62
63              fprintf(fid, '\n');
64
65              fprintf(fid, '%s\n', sprintf('%s %d', kcenters, ...
                    center_count));
66              fprintf(fid, '%s\n', sprintf('%s %s', run_kmeans, ...
                    char(heuristics(h))));
67          end
68
69          fprintf(fid, '%s\n', quit);
70
71          fclose(fid);
72      end
73  end
```

```
1   function rems = gen_rems(spectrum, n_nodes, dim, thresh, bins)
2       rems = ones(n_nodes, bins);
3       %aggregate = ones(1, bins);
4       rem = zeros(1, bins);
5       %fid = fopen(filename, 'w');
6       rem_count = 1;
7
8       %if fid == -1
9       %    error('could not open file');
10      %end
11
12      for x = 1:dim
13          for y = 1:dim
14              cell = spectrum(x, y, :);
15              c = cell(:);
16              thresh = mean(cell);
17
18              if c == zeros(bins, 1)
19                  rem = ones(1, bins);
20              else
21                  for i = 1:bins
22                      if c(i) < thresh
23                          rem(i) = 1;
24                      else
25                          rem(i) = 0;
26                      end
27                  end
28              end
29
30              rems(rem_count, :) = rem;
31      %        aggregate = aggregate & rem;
32      %        fprintf(fid, '%s\n', dec2hex(rem)');
```

```
            rem_count = rem_count + 1;
        end
    end

    %fprintf(fid, '\n%s\n', dec2hex(aggregate)');

    %openChannels = 0;

    %for i = 1:bins
    %    if aggregate(i) == 1
    %        openChannels = openChannels + 1;
    %    end
    %end

    %fprintf('Open Channels: %d\n', openChannels);

    %fclose(fid);
end
```

```
function maps = gen_test_maps(n_maps, size, dim)
    maps = zeros(n_maps, 3, size);

    for i = 1:n_maps
        maps(i, 1:2, :) = gen_data_pts(size, dim, ...
            sprintf('./input/input%d', i))';
        maps(i, 3, :) = 0;
    end
end
```

```
function kmeans_experiment(n_trials, n_nodes, v_n_centers, ...
    thresh_adjust, map_dimensions, bins, spectrum_number, ...
    distro_indices, baseline, plot_clusters)
    lloyd = zeros(numel(v_n_centers), numel(distro_indices));
    swap = zeros(numel(v_n_centers), numel(distro_indices));
    hybrid = zeros(numel(v_n_centers), numel(distro_indices));
    ez_hybrid = zeros(numel(v_n_centers), numel(distro_indices));

    n_heuristics = 4;
    n_centers = numel(v_n_centers);

    for t = 0.6:0.1:1.4
    thresh_adjust = t;

    % Initialize spectrum data if it does not already exist
    fprintf('Generating spectrum data using Sim%d_Grid.mat ...', ...
        spectrum_number);
    spectrum = load_spectrum(spectrum_number);
    fprintf('Done.\n');

```

```
18
19      % Point distribution types
20      distros = {'uniform' 'gauss' 'multi_clus'};%; 'co_gauss';
21                 %'co_laplace'; 'clus_gauss'; 'clus_orth_flats';
22                 %'clus_ellipsoids'; 'multi_clus'];
23
24      tic
25      rems = gen_rems(spectrum, n_nodes, map_dimensions, ...
           thresh_adjust, bins);
26
27      fclose all;
28      clean_data();
29
30  %    icss_figure_number = numel(distro_indices)*n_trials;
31
32  %    for j = 1:numel(distro_indices)
33      for j = distro_indices
34          seed_number = 0;
35
36          intra_cluster_similarity = zeros(n_trials, ...
               numel(v_n_centers), 4);
37
38          fprintf('Running experiment using %s distribution...\n', ...
               char(distros(j)));
39
40  %        for i = 1:n_trials
41          for i = n_trials
42              seed_number = seed_number + 1;
43
44              % When seed is 3, kmltest.exe crashes. (??)
45              if seed_number == 3
46                  seed_number = 4;
47              end
48
49              fprintf('Generating K-means commands...');
50              for v = v_n_centers
51                  gen_kmeans_commands(i, n_nodes, v_n_centers, j, ...
                       v, seed_number, baseline);
52              end
53              fprintf('Done.\n');
54
55              fprintf('Running kmltest.exe for map %d...', i);
56              for v = v_n_centers
57                  run_kmeans(i, v);
58              end
59              fprintf('Done.\n');
60
61              fprintf('Parsing points for this distribution...');
62              points = read_kmeans_points(i, n_nodes, ...
                   map_dimensions, baseline);
63              fprintf('Done.\n');
64
```

```
65              fprintf('Parsing assignments for map %d...', i);
66              assignments = read_kmeans_assignments(i, n_nodes, ...
                    numel(v_n_centers));
67              fprintf('Done.\n');
68
69              fprintf('Parsing centers for map %d...', i);
70              centers = read_kmeans_centers(i, n_nodes, ...
                    numel(v_n_centers));
71              fprintf('Done.\n');
72
73              fprintf('Entering assignments for map %d...', i);
74              map_assigned = set_assignments(n_nodes, points, ...
                    assignments, v_n_centers);
75              fprintf('Done.\n');
76
77              % Could use FFT algorithm for faster similarity ...
                    comparison
78
79              fprintf('Drawing clusters and determining ...
                    intra-cluster similarity...');
80              %figure((j-1)*n_trials + i)
81              %colorbar
82              %colorbar('YTickLabel', {'0%', '25%', '50%', '75%', ...
                    '100%'}, ...
83              %         'location', 'EastOutside')
84              temp = draw_clusters(i, j, v_n_centers, map_assigned, ...
                    n_nodes, rems, baseline, plot_clusters);
85              intra_cluster_similarity(i, :, :) = temp;
86              fprintf('Done.\n');
87
88              fprintf('Map %d processing complete!\n', i);
89              fprintf('=======================================\n');
90          end
91
92          for k = 1:numel(v_n_centers)
93              lloyd(k, j) = mean(intra_cluster_similarity(:, k, 1));
94              swap(k, j) = mean(intra_cluster_similarity(:, k, 2));
95              hybrid(k, j) = mean(intra_cluster_similarity(:, k, 3));
96              ez_hybrid(k, j) = mean(intra_cluster_similarity(:, k, ...
                    4));
97          end
98
99  %             fprintf('Plotting intra-cluster similarity for %s ...
        distribution...', char(distros(j)));
100 %             %intra_cluster_similarity
101 %
102 %             figure(n_trials*numel(distro_indices) + j)
103 %             hold on
104 %             plot(log2(v_n_centers), intra_cluster_similarity(:, ...
        :, 1), '--r+')
105 %             plot(log2(v_n_centers), intra_cluster_similarity(:, ...
        :, 2), '--b+')
```

```
106 %             plot(log2(v_n_centers), intra_cluster_similarity(:, ...
    :, 3), '--k+')
107 %             plot(log2(v_n_centers), intra_cluster_similarity(:, ...
    :, 4), '--g+')
108 %             set(gca, 'XTickLabel', v_n_centers)
109 %             xlabel('Number of Clusters')
110 %             ylabel('Average ICSS')
111 %             title(sprintf('ICSS vs. Cluster Count for %s ...
    Distribution', upper(char(distros(j)))));
112 %             hold off
113 %             fprintf('Done.\n');
114
115 %           fprintf('Experiment complete for %s ...
    distribution!\n', char(distros(j)));
116 %           fprintf('========================================\n\n');
117
118 %             colors = ['r' 'b' 'g' 'k'];
119
120 %         Plot ICSS for Lloyd
121
122         fprintf('Plotting ICSS for all heuristics...');
123         figure(n_heuristics*n_centers + 1)
124         h = zeros(4, 1);
125         labels = {'Lloyd', 'Swap', 'Hybrid', 'EZ-Hybrid'};
126
127         hold on
128         n_icss = n_trials*numel(v_n_centers);
129         cluster = zeros(n_icss, 1);
130         icss = zeros(n_icss, 1);
131
132         for m = 1:numel(v_n_centers)
133             for n = 1:n_trials
134                 cluster((m-1)*n_trials + n) = v_n_centers(m);
135                 icss((m-1)*n_trials + n) = ...
                        intra_cluster_similarity(n, m, 1);
136             end
137         end
138
139 %         boxplot(icss, log(cluster))
140
141         h(1) = plot(v_n_centers, lloyd(:, j), '-ro');
142 %         h(2) = plot(v_n_centers, swap(:, j), '-bs');
143 %         h(3) = plot(v_n_centers, hybrid(:, j), '-g*');
144 %         h(4) = plot(v_n_centers, ez_hybrid(:, j), '-cv');
145
146         set(gcf,'Color','white')
147         set(gca, 'XTickMode', 'auto')
148         set(gca, 'XTickLabelMode', 'auto')
149         set(gca,'Box','on')
150         set(gca,'XMinorTick', 'on')
151         set(gca,'YMinorTick', 'on')
152         axis([0 n_nodes 0.6 1]);
```

```
153
154         xlabel('Number of Clusters')
155         ylabel('Average ICSS')
156         title(sprintf('ICSS vs. Cluster Count for %d ...
               Nodes\n(using Sim%d_Grid.mat, %s distribution)', ...
               n_nodes, spectrum_number, char(distros(j))), ...
               'Interpreter', 'none');
157         legend(h, labels, 'Location', 'NorthWest');
158         hold off
159         fprintf('Done.\n');
160         export_fig(gcf, ...
               sprintf('./figures/ICSS/icss_sim%d_%s.png', ...
               spectrum_number, char(distros(j))));
161 %        close(gcf);
162
163 %         icss_figure_number = icss_figure_number + 1;
164 %
165 %         Plot ICSS for Swap
166
167 %        fprintf('Plotting ICSS for Swap...');
168 %             figure(icss_figure_number)
169 %             h_swap = zeros(numel(thresholds), 1);
170 %             labels = cell(numel(thresholds), 1);
171 %
172 %         hold on
173 %         n_icss = n_trials*numel(v_n_centers);
174 %         cluster = zeros(n_icss, 1);
175 %         icss = zeros(n_icss, 1);
176 %
177 %         for m = 1:numel(v_n_centers)
178 %             for n = 1:n_trials
179 %                 cluster((m-1)*n_trials + n) = v_n_centers(m);
180 %                 icss((m-1)*n_trials + n) = ...
    intra_cluster_similarity(n, m, 2);
181 %             end
182 %         end
183
184 %         boxplot(icss, log(cluster))
185
186 %             for r = 1:numel(thresholds)
187 %                 h_swap(r) = plot(v_n_centers, swap(r, :, j), ...
    '-bo');
188 %                 labels(r) = {char(sprintf('Threshold @ %d dB', ...
    thresholds(r)))};
189 %             end
190
191 %         axis ([0.90*log2(min(v_n_centers)) ...
    1.05*log2(max(v_n_centers)) ...
192 %           0.95*min(min(min(lloyd), min(swap)), ...
    min(min(hybrid), ...
193 %                    min(ez_hybrid))) ...
```

```
194 %           1.05*max(max(max(lloyd), max(swap)), ...
    max(max(hybrid), ...
195 %                   max(ez_hybrid)))])
196 %
197 %           set(gca, 'XTickMode', 'auto')
198 %           set(gca, 'XTickLabel', v_n_centers)
199 %
200 %           xlabel('Number of Clusters')
201 %           ylabel('Average ICSS')
202 %           title(sprintf('ICSS vs. Cluster Count for Swap ...
    Heuristic\n(%s distribution)', char(distros(j))));
203 %           legend(h_swap, labels, 'Location', 'SouthEast');
204 %           hold off
205 %           fprintf('Done.\n');
206 %
207 %       icss_figure_number = icss_figure_number + 1;
208 %
209 %       Plot ICSS for Hybrid
210 %
211 %           fprintf('Plotting ICSS for Hybrid...');
212 %           figure(icss_figure_number)
213 %           h_hybrid = zeros(numel(thresholds), 1);
214 %           labels = cell(numel(thresholds), 1);
215 %
216 %           hold on
217 %           n_icss = n_trials*numel(v_n_centers);
218 %           cluster = zeros(n_icss, 1);
219 %           icss = zeros(n_icss, 1);
220 %
221 %           for m = 1:numel(v_n_centers)
222 %               for n = 1:n_trials
223 %                   cluster((m-1)*n_trials + n) = v_n_centers(m);
224 %                   icss((m-1)*n_trials + n) = ...
    intra_cluster_similarity(n, m, 3);
225 %               end
226 %           end
227 %
228 %       boxplot(icss, log(cluster))
229 %
230 %           h_hybrid = plot(v_n_centers, hybrid(r, :, j), '-go');
231 %           labels(r) = {char(sprintf('Threshold @ %d dB', ...
    thresholds(r)))};
232 %
233 %       axis ([0.90*log2(min(v_n_centers)) ...
    1.05*log2(max(v_n_centers)) ...
234 %           0.95*min(min(min(lloyd), min(swap)), ...
    min(min(hybrid), ...
235 %                   min(ez_hybrid))) ...
236 %           1.05*max(max(max(lloyd), max(swap)), ...
    max(max(hybrid), ...
237 %                   max(ez_hybrid)))])
238
```

```
239 %             set(gca, 'XTickMode', 'auto')
240 %             set(gca, 'XTickLabel', v_n_centers)
241 %
242 %             xlabel('Number of Clusters')
243 %             ylabel('Average ICSS')
244 %             title(sprintf('ICSS vs. Cluster Count for Hybrid ...
    Heuristic\n(%s distribution)', char(distros(j))));
245 %             legend(h_hybrid, labels, 'Location', 'SouthEast');
246 %             hold off
247 %             fprintf('Done.\n');
248
249 %         icss_figure_number = icss_figure_number + 1;
250 %
251 %         Plot ICSS for EZ-Hybrid
252
253 %             fprintf('Plotting ICSS for EZ-Hybrid...');
254 %             figure(icss_figure_number)
255 %             h_ez_hybrid = zeros(numel(thresholds), 1);
256 %             labels = cell(numel(thresholds), 1);
257 %
258 %             hold on
259 %             n_icss = n_trials*numel(v_n_centers);
260 %             cluster = zeros(n_icss, 1);
261 %             icss = zeros(n_icss, 1);
262 %
263 %             for m = 1:numel(v_n_centers)
264 %                 for n = 1:n_trials
265 %                     cluster((m-1)*n_trials + n) = v_n_centers(m);
266 %                     icss((m-1)*n_trials + n) = ...
    intra_cluster_similarity(n, m, 4);
267 %                 end
268 %             end
269 %
270 %             %boxplot(icss, log(cluster))
271 %
272 %             for r = 1:numel(thresholds)
273 %                 h_ez_hybrid(r) = plot(v_n_centers, ez_hybrid(r, ...
    :, j), '-ko');
274 %                 labels(r) = {char(sprintf('Threshold @ %d dB', ...
    thresholds(r)))};
275 %             end
276 %
277 %             set(gca, 'XTickMode', 'auto')
278 %             set(gca, 'XTickLabel', v_n_centers)
279 %
280 %             xlabel('Number of Clusters')
281 %             ylabel('Average ICSS')
282 %             title(sprintf('ICSS vs. Cluster Count for EZ-Hybrid ...
    Heuristic\n(%s distribution)', char(distros(j))));
283 %             legend(h_ez_hybrid, labels, 'Location', 'SouthEast');
284 %             hold off
285 %             fprintf('Done.\n');
```

```
286
287 %             icss_figure_number = icss_figure_number + 1;
288     end
289     toc
290
291     % Catch any file handles not closed due to an error
292     fclose all;
293
294     end
295 end
```

```
1  function spectrum = load_spectrum(sim_number)
2      load(sprintf('Sim%d_Grid.mat', sim_number), ...
3          'device_grid', 'freq', 'info_grid', 'mag');
4      spectrum = decode_spectrum(mag, 7, info_grid, 2048, 10);
5  end
```

```
1  function plot_spectrum(map_number, spectrum, thresh_adjust, bins)
2      figure(map_number);
3
4      title(sprintf('RF Spectrum Maps Generated by ...
           DYSE\nSim%d_Grid.mat', map_number))
5
6      plot_number = 1;
7      unknown = 0;
8
9      colors = ['r' 'b' 'g' 'k'];
10
11     for y = 1:10
12     %for y = 4
13         for x = 1:10
14         %for x = 3
15             set(gca, 'Units', 'normal');
16             subaxis(10, 10, plot_number, 'Spacing', 0.015, 'MT', ...
17                     0.03, 'MR', 0.02, 'MB', 0.035, 'ML', 0.02);
18             title(sprintf('%d, %d', x, y))
19
20             s = spectrum(x, y, :);
21             threshold = thresh_adjust*mean(s(:));
22
23             plot(s(:));
24             axis([0 2047 -150 10]);
25             hold on
26
27             plot([0 bins], [threshold threshold], '--', 'Color', ...
                   (1/255)*[5, 136, 5], 'LineWidth', 2);
28             plot([0 bins], [mean(s(:)) mean(s(:))], '--', ...
                   'Color', [1, 0, 0], 'LineWidth', 2);
29
```

```
            hold off
            set(gca, 'xtick', []);
            set(gca, 'ytick', []);

            if x == 1
                set(gca, 'YTickMode', 'manual')
                set(gca, 'YTick', [-120, -60, 0])
            elseif x == 10
                set(gca, 'YTickMode', 'manual')
                set(gca, 'YTick', [-120, -60, 0])
                set(gca, 'YAxisLocation', 'right')
            end

            if y == 1
                set(gca, 'XTickMode', 'manual')
                set(gca, 'XTick', [0 1023 2047])
                set(gca, 'XAxisLocation', 'top')
            elseif y == 10
                set(gca, 'XTickMode', 'manual')
                set(gca, 'XTick', [0 1023 2047])
            end

            plot_number = plot_number + 1;
        end
    end

    fprintf('%d total cells with unknown properties.\n\n', unknown);

    set(gcf, 'Color', 'white')
    set(gcf, 'Position', [1 1 1920 978])

    export_fig(gcf, sprintf('.\\figures\\RF_maps\\dyse%d.png', ...
        map_number));
end
```

```
function assignments = read_kmeans_assignments(map_number, ...
    node_count, n_centers)
    lloyd = 1;
    swap = 2;
    hybrid = 3;
    EZhybrid = 4;
    type = 0;

    n_heuristics = 4;

    centers_count = -1;
    centers_index = 0;

    assignments = zeros(n_centers, n_heuristics, node_count);

    fid = fopen(sprintf('./output/output%d', map_number), 'r');
```

```
16
17      while ¬feof(fid)
18          % Spin until cluster assignments
19          while (strcmp(fgetl(fid), '[Run_k-means:') == 0)
20              if feof(fid)
21                  break;
22              end
23          end
24
25          % Get the heuristic type
26          str_type = fscanf(fid, '  k-means_alg      = %s', 1);
27          if strcmp(str_type, 'lloyd') ≠ 0
28              type = lloyd;
29          elseif strcmp(str_type, 'swap') ≠ 0
30              type = swap;
31          elseif strcmp(str_type, 'hybrid') ≠ 0
32              type = hybrid;
33          elseif strcmp(str_type, 'EZ-hybrid') ≠ 0
34              type = EZhybrid;
35          elseif strcmp(str_type, '') ≠ 0
36              %disp('Reached end of output file.');
37              % Reached end of file => no more results to read
38              break;
39          end
40
41          % Read out the next line
42          fgetl(fid);
43          fgetl(fid);
44
45          % Get the number of centers
46          temp_centers = fscanf(fid, '  kcenters         = %d', 1);
47          if centers_count ≠ temp_centers
48              centers_count = temp_centers;
49              centers_index = centers_index + 1;
50          end
51
52          % Spin until cluster assignments
53          while (strcmp(fgetl(fid), '  (Cluster assignments:') == 0)
54              if feof(fid)
55                  break;
56              end
57          end
58
59          % Read out two following lines
60          fgetl(fid);
61          fgetl(fid);
62
63          for i = 1:node_count
64              % Read the cluster for the point number
65              data = fscanf(fid, '%d %d %f', 3)';
66
67              if(isempty(data))
```

```
                break;
            end

            assignments(centers_index, type, i) = data(2);
        end
    end

    fclose(fid);
end
```

```
function center_data = read_kmeans_centers(map_number, ...
    node_count, n_centers)
    lloyd = 1;
    swap = 2;
    hybrid = 3;
    EZhybrid = 4;
    type = 0;

    n_heuristics = 4;

    centers_count = -1;
    centers_index = 0;

    center_data = zeros(n_centers, n_heuristics, node_count, 2);

    fid = fopen(sprintf('./output/output%d', map_number), 'r');

    while ¬feof(fid)
        % Spin until cluster assignments
        while (strcmp(fgetl(fid), '[Run_k-means:') == 0)
            if feof(fid)
                break;
            end
        end

        % Get the heuristic type
        str_type = fscanf(fid, '  k-means_alg      = %s', 1);
        if strcmp(str_type, 'lloyd') ≠ 0
            type = lloyd;
        elseif strcmp(str_type, 'swap') ≠ 0
            type = swap;
        elseif strcmp(str_type, 'hybrid') ≠ 0
            type = hybrid;
        elseif strcmp(str_type, 'EZ-hybrid') ≠ 0
            type = EZhybrid;
        elseif strcmp(str_type, '') ≠ 0
            %disp('Reached end of output file.');
            % Reached end of file => no more results to read
            break;
        end

```

```
        % Read out the next line
        fgetl(fid);
        fgetl(fid);

        % Get the number of centers
        temp_centers = fscanf(fid, '  kcenters         = %d', 1);
        if centers_count ~= temp_centers
            centers_count = temp_centers;
            centers_index = centers_index + 1;
        end

        % Spin until center points
        while (strcmp(fgetl(fid), '  (Final Center Points:') == 0)
            if feof(fid)
                break;
            end
        end

        for i = 1:centers_count
            temp_center = fscanf(fid, '       %d    [    %f    %f ...
                ] dist =    %f', 4)';

            center = temp_center(1) + 1;
            center_x = temp_center(2);
            center_y = temp_center(3);
            center_sqdist = temp_center(4);

            center_data(centers_index, type, center, 1:2) = ...
                [center_x center_y];
        end
    end

    fclose(fid);
end
```

```
function points = read_kmeans_points(map_number, node_count, dim, ...
    baseline)
    points = zeros(node_count, 3);

    fid1 = 0;
    if baseline == 0
        fid1 = fopen(sprintf('./output/output%d', map_number), 'r');
    else
        fid1 = fopen(sprintf('./input/baseline%d', baseline), 'r');
    end

    fid2 = fopen(sprintf('./input/input%d', map_number), 'w');

    i = 0;
    x = 0;
    y = 0;
```

```
16
17      if baseline == 0
18          while ¬feof(fid1)
19              % Spin until cluster assignments
20              while (strcmp(fgetl(fid1), '  (Data_Points:') == 0)
21                  if feof(fid1)
22                      break;
23                  end
24              end
25
26              for i = 1:node_count
27                  if feof(fid1)
28                      break;
29                  end
30
31                  % Get the point coordinates
32                  temp_point = fscanf(fid1, '    %d   [  %f  %f ]', ...
                        3)';
33
34                  x = (temp_point(2)+1)*(dim/2);
35                  y = (temp_point(3)+1)*(dim/2);
36
37                  fprintf(fid2, '%f %f\n', x, y);
38
39                  points(temp_point(1)+1, 1) = x;
40                  points(temp_point(1)+1, 2) = y;
41              end
42          end
43      else
44          for i = 1:node_count
45              if feof(fid1)
46                  break;
47              end
48
49              temp_point = fscanf(fid1, '%f %f', 2)';
50
51              x = temp_point(1);
52              y = temp_point(2);
53
54              fprintf(fid2, '%f %f\n', x, y);
55
56              points(i, 1) = x;
57              points(i, 2) = y;
58          end
59      end
60
61      fclose(fid1);
62      fclose(fid2);
63  end
```

```
1  function run_kmeans(map_number, center_count)
```

```
    heuristics = {'lloyd', 'swap', 'hybrid', 'EZ-hybrid'};

    output = sprintf('./output/output%d', map_number);

    for h = 1:numel(heuristics)
        input = sprintf('./commands/commands%d_%d_%s', ...
            map_number, center_count, char(heuristics(h)));

        dos(sprintf('kmltest.exe < %s >> %s', input, output));
    end
end
```

```
function map_assigned = set_assignments(n_nodes, map, ...
    assignments, v_n_centers)
    n_heuristics = 4;

    map_assigned = zeros(numel(v_n_centers), n_heuristics, 3, ...
        n_nodes);

    for j = 1:numel(v_n_centers)
        for i = 1:n_heuristics
            temp_map = map;
            temp_assignments = assignments(j, i, :);
            temp_map(:,3) = temp_assignments(:);

%             temp_map(:, :)
%             temp_map_assigned = map_assigned(j, i, :, :)
%             map_assigned(j, i, :, :) = temp_map(:, :);

            for k = 1:n_nodes
                map_assigned(j, i, 1, k) = temp_map(k, 1);
                map_assigned(j, i, 2, k) = temp_map(k, 2);
                map_assigned(j, i, 3, k) = temp_map(k, 3);
            end
        end
    end
end
```

Appendix G: VHDL Code

VHDL files represent the FPGA-based circuitry used to quickly generate an adaptive hopset. The overall hopset selector is wired together in `adaptive_hopset_selector.vhd`, and actual hopset selection is performed in `hopset_selector_b.vhd`. Files beginning with "`tb_`" are test bench files for verifying system and subsystem functionality. A total of 1,430 lines of VHDL code were implemented for this research.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity adaptive_hopset_selector is
6    port(
7      signal clk, reset       : in std_logic;
8      signal load_rem         : in std_logic;
9      signal load_key         : in std_logic;
10     signal new_rems         : in std_logic;
11     signal new_key          : in std_logic;
12     signal input            : in std_logic_vector(31 downto 0);
13     signal key              : in std_logic_vector(31 downto 0);
14     signal lower_bw_mask    : in std_logic_vector(31 downto 0);
15     signal upper_bw_mask    : in std_logic_vector(31 downto 0);
16     signal start            : in std_logic;
17     signal next_hop         : in std_logic;
18     signal hopset_ready     : out std_logic;
19     signal finished         : out std_logic;
20     signal hop_number       : out std_logic_vector(10 downto 0);
21     signal hop_channel      : out std_logic_vector(10 downto 0)
22   );
23 end entity;
24
25 architecture structural of adaptive_hopset_selector is
26   component rem_merger is
27     port(
28       signal clk, reset     : in std_logic;
29       signal load_rem       : in std_logic;
30       signal new_rems       : in std_logic;
31       signal input          : in std_logic_vector(31 downto 0);
32       signal rem_rdat       : in std_logic_vector(31 downto 0);
33       signal rem_rsel       : out std_logic_vector(5 downto 0);
34       signal rem_wen        : out std_logic;
35       signal rem_wsel       : out std_logic_vector(5 downto 0);
36       signal rem_wdat       : out std_logic_vector(31 downto 0)
37     );
```

```
38    end component;
39
40    component key_loader is
41      port(
42        signal clk, reset     : in std_logic;
43        signal load_key       : in std_logic;
44        signal new_key        : in std_logic;
45        signal input          : in std_logic_vector(31 downto 0);
46        signal key_wen        : out std_logic;
47        signal key_wsel       : out std_logic_vector(5 downto 0);
48        signal key_wdat       : out std_logic_vector(31 downto 0)
49      );
50    end component;
51
52    component hopset_selector_b is
53      port(
54        signal clk, reset     : in std_logic;
55        signal lower_bw_mask  : in std_logic_vector(31 downto 0);
56        signal upper_bw_mask  : in std_logic_vector(31 downto 0);
57        signal start          : in std_logic;
58        signal next_hop       : in std_logic;
59        signal new_rems       : in std_logic;
60        signal new_key        : in std_logic;
61        signal hopset_ready   : out std_logic;
62        signal finished       : out std_logic;
63        signal hop_number     : out std_logic_vector(10 downto 0);
64        signal hop_channel    : out std_logic_vector(10 downto 0);
65
66        signal rem_rdat       : in std_logic_vector(31 downto 0);
67        signal rem_rsel       : out std_logic_vector(5 downto 0);
68
69        signal key_rdat       : in std_logic_vector(31 downto 0);
70        signal key_rsel       : out std_logic_vector(5 downto 0);
71
72        signal bram_write_en  : buffer std_logic := '0';
73        signal bram_read_en   : out std_logic := '0';
74        signal bram_fill      : out std_logic;
75        signal bram_addr      : buffer std_logic_vector(10 downto ...
              0) := (others => '0');
76        signal bram_data_in   : out std_logic_vector(10 downto 0) ...
              := (others => '0');
77        signal bram_data_out  : in std_logic_vector(10 downto 0) := ...
              (others => '0')
78      );
79    end component;
80
81    component bram_shared_2port is
82      port(
83          signal clk, reset   :   in  std_logic;
84          signal wdat         :   in  std_logic_vector (31 downto 0);
85          signal wsel         :   in  std_logic_vector (5 downto 0);
86          signal wen          :   in  std_logic;
```

```
87          signal rsel1        :   in  std_logic_vector (5 downto 0);
88          signal rsel2        :   in  std_logic_vector (5 downto 0);
89          signal rdat1        :   out std_logic_vector (31 downto 0);
90          signal rdat2        :   out std_logic_vector (31 downto 0)
91      );
92    end component;
93
94    component bram_channels_table is
95      port(
96          signal clka, rsta   : in std_logic;
97          signal wea          : in std_logic_vector(0 downto 0);
98          signal ena : in std_logic;
99          signal regcea : in std_logic;
100         signal addra        : in std_logic_vector(10 downto 0);
101         signal dina         : in std_logic_vector(10 downto 0);
102         signal douta        : out std_logic_vector(10 downto 0)
103     );
104   end component;
105
106     signal rem_wdat         : std_logic_vector (31 downto 0);
107     signal rem_wsel         : std_logic_vector (5 downto 0);
108     signal rem_wen          : std_logic;
109     signal rem_rsel1        : std_logic_vector (5 downto 0);
110     signal rem_rsel2        : std_logic_vector (5 downto 0);
111     signal rem_rdat1        : std_logic_vector (31 downto 0);
112     signal rem_rdat2        : std_logic_vector (31 downto 0);
113
114     signal key_wdat         : std_logic_vector (31 downto 0);
115     signal key_wsel         : std_logic_vector (5 downto 0);
116     signal key_wen          : std_logic;
117     signal key_rsel1        : std_logic_vector (5 downto 0);
118     signal key_rsel2        : std_logic_vector (5 downto 0);
119     signal key_rdat1        : std_logic_vector (31 downto 0);
120     signal key_rdat2        : std_logic_vector (31 downto 0);
121
122     signal bram_write_en    : std_logic_vector(0 downto 0) := ...
          (others => '0');
123     signal bram_read_en     : std_logic := '0';
124     signal bram_fill        : std_logic := '0';
125     signal bram_addr        : std_logic_vector(10 downto 0) := ...
          (others => '0');
126     signal bram_data_in     : std_logic_vector(10 downto 0) := ...
          (others => '0');
127     signal bram_data_out    : std_logic_vector(10 downto 0) := ...
          (others => '0');
128 begin
129   key_rsel2 <= (others => 'Z');
130   key_rdat2 <= (others => 'Z');
131
132   merger: rem_merger
133   port map(
134     clk => clk,
```

```
135       reset => reset,
136       load_rem => load_rem,
137       new_rems => new_rems,
138       input => input,
139       rem_rdat => rem_rdat1,
140       rem_rsel => rem_rsel1,
141       rem_wen => rem_wen,
142       rem_wsel => rem_wsel,
143       rem_wdat => rem_wdat
144     );
145
146     loader: key_loader
147     port map(
148       clk => clk,
149       reset => reset,
150       load_key => load_key,
151       new_key => new_key,
152       input => key,
153       key_wen => key_wen,
154       key_wsel => key_wsel,
155       key_wdat => key_wdat
156     );
157
158     selector: hopset_selector_b
159     port map(
160       clk => clk,
161       reset => reset,
162       lower_bw_mask => lower_bw_mask,
163       upper_bw_mask => upper_bw_mask,
164       start => start,
165       next_hop => next_hop,
166       new_rems => new_rems,
167       new_key => new_key,
168       bram_fill => bram_fill,
169       hopset_ready => hopset_ready,
170       finished => finished,
171       hop_number => hop_number,
172       hop_channel => hop_channel,
173
174       rem_rdat => rem_rdat2,
175       rem_rsel => rem_rsel2,
176
177       key_rdat => key_rdat1,
178       key_rsel => key_rsel1,
179
180       bram_write_en => bram_write_en(0),
181       bram_read_en => bram_read_en,
182       bram_addr => bram_addr,
183       bram_data_in => bram_data_in,
184       bram_data_out => bram_data_out
185     );
186
```

```
187   bram_rem: bram_shared_2port
188   port map(
189     clk => clk,
190     reset => new_rems,
191     wdat => rem_wdat,
192     wsel => rem_wsel,
193     wen => rem_wen,
194     rsel1 => rem_rsel1,
195     rsel2 => rem_rsel2,
196     rdat1 => rem_rdat1,
197     rdat2 => rem_rdat2
198   );
199
200   bram_key: bram_shared_2port
201   port map(
202     clk => clk,
203     reset => new_key,
204     wdat => key_wdat,
205     wsel => key_wsel,
206     wen => key_wen,
207     rsel1 => key_rsel1,
208     rsel2 => key_rsel2,
209     rdat1 => key_rdat1,
210     rdat2 => key_rdat2
211   );
212
213   bram: bram_channels_table
214   port map(
215     clka => clk,
216     rsta => new_rems,
217     wea => bram_write_en,
218     ena => bram_read_en,
219     regcea => bram_fill,
220     addra => bram_addr,
221     dina => bram_data_in,
222     douta => bram_data_out
223   );
224 end structural;
```

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use ieee.std_logic_unsigned.all;
5
6  -- entity bram_channels_table is
7    -- port(
8      -- signal clk, reset : in std_logic;
9      -- signal write_en : in std_logic;
10     -- signal read_en : in std_logic;
11     -- signal fill : in std_logic;
12     -- signal addr : in integer;
```

```
13      -- signal data_in : in std_logic_vector(10 downto 0);
14      -- signal data_out : out std_logic_vector(10 downto 0)
15    -- );
16  -- end entity bram_channels_table;
17
18  entity bram_channels_table is
19    port (
20      clka : in std_logic;                          -- clk
21      rsta : in std_logic;                          -- reset
22      ena : in std_logic;                           -- enable
23      regcea : in std_logic;                        -- fill
24      wea : in std_logic_vector(0 downto 0);        -- write enable
25      addra : in std_logic_vector(10 downto 0);     -- address
26      dina : in std_logic_vector(10 downto 0);      -- data in
27      douta : out std_logic_vector(10 downto 0)     -- data out
28    );
29  end bram_channels_table;
30
31  architecture structural of bram_channels_table is
32    type channel_list is array (0 to 2048) of std_logic_vector(10 ...
        downto 0);
33    signal channels_table : channel_list;
34
35    signal open_channels : integer := 0;
36
37    signal fill : std_logic := '0';
38    signal fill_table : std_logic := '0';
39    signal fill_start : std_logic := '0';
40    signal low_offset : integer := 0;
41    signal high_offset : integer := 0;
42    signal increment : integer := 0;
43
44    signal write_en_buffer : std_logic := '0';
45    signal read_en_buffer : std_logic := '0';
46  begin
47    channels_table(2048) <= (others => '0');
48    fill <= regcea;
49
50    bram: process(clka, rsta, wea, ena, fill)
51    begin
52      if(rsta = '1') then
53        write_en_buffer <= '0';
54        read_en_buffer <= '0';
55        open_channels <= 0;
56        douta <= (others => '0');
57      elsif(rising_edge(clka)) then
58        write_en_buffer <= wea(0);
59        read_en_buffer <= ena;
60        open_channels <= open_channels;
61
62        if(wea(0) = not write_en_buffer) then
```

```
63          channels_table(to_integer(unsigned(addra))) <= dina(10 ...
               downto 0);
64          open_channels <= open_channels + 1;
65        end if;
66      elsif(fill = '1') then
67        fill_table <= '1';
68        fill_start <= '1';
69      elsif(fill_table = '1') then
70        if(fill_start = '1') then
71          low_offset <= open_channels;
72          high_offset <= open_channels+open_channels-1;
73          increment <= open_channels;
74          fill_start <= '0';
75        elsif(fill_start = '0' and fill_table = '1') then
76          if(high_offset < 2047) then
77            channels_table(low_offset to high_offset) <= ...
                  channels_table(0 to open_channels-1);
78            low_offset <= low_offset + increment;
79            high_offset <= high_offset + increment;
80          else
81            fill_table <= '0';
82            channels_table(low_offset to 2047) <= channels_table(0 ...
                  to (2047-low_offset));
83          end if;
84        else
85          low_offset <= 0;
86          high_offset <= 0;
87          increment <= 0;
88          fill_table <= '0';
89        end if;
90      elsif(rising_edge(ena) and fill_table = '0') then
91        douta <= channels_table(to_integer(unsigned(addra)))(10 ...
             downto 0);
92      end if;
93    end process bram;
94  end structural;
```

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity bram_shared_2port is
5    port(
6          signal clk, reset   :   in  std_logic;
7          signal wdat :   in  std_logic_vector (31 downto 0);
8          signal wsel :   in  std_logic_vector (5 downto 0);
9          signal wen  :   in  std_logic;
10         signal rsel1    :   in  std_logic_vector (5 downto 0);
11         signal rsel2    :   in  std_logic_vector (5 downto 0);
12         signal rdat1    :   out std_logic_vector (31 downto 0);
13         signal rdat2    :   out std_logic_vector (31 downto 0)
14   );
```

```
15 end entity;
16
17 architecture structural of bram_shared_2port is
18     type shmem64x32 is array (0 to 63) of std_logic_vector(31 ...
          downto 0);
19     signal mem  :   shmem64x32;
20 begin
21   shmem: process(clk, reset)
22   begin
23     if(reset = '1') then
24       mem(0) <= (others => '1');
25       mem(1) <= (others => '1');
26       mem(2) <= (others => '1');
27       mem(3) <= (others => '1');
28       mem(4) <= (others => '1');
29       mem(5) <= (others => '1');
30       mem(6) <= (others => '1');
31       mem(7) <= (others => '1');
32       mem(8) <= (others => '1');
33       mem(9) <= (others => '1');
34       mem(10) <= (others => '1');
35       mem(11) <= (others => '1');
36       mem(12) <= (others => '1');
37       mem(13) <= (others => '1');
38       mem(14) <= (others => '1');
39       mem(15) <= (others => '1');
40       mem(16) <= (others => '1');
41       mem(17) <= (others => '1');
42       mem(18) <= (others => '1');
43       mem(19) <= (others => '1');
44       mem(20) <= (others => '1');
45       mem(21) <= (others => '1');
46       mem(22) <= (others => '1');
47       mem(23) <= (others => '1');
48       mem(24) <= (others => '1');
49       mem(25) <= (others => '1');
50       mem(26) <= (others => '1');
51       mem(27) <= (others => '1');
52       mem(28) <= (others => '1');
53       mem(29) <= (others => '1');
54       mem(30) <= (others => '1');
55       mem(31) <= (others => '1');
56       mem(32) <= (others => '1');
57       mem(33) <= (others => '1');
58       mem(34) <= (others => '1');
59       mem(35) <= (others => '1');
60       mem(36) <= (others => '1');
61       mem(37) <= (others => '1');
62       mem(38) <= (others => '1');
63       mem(39) <= (others => '1');
64       mem(40) <= (others => '1');
65       mem(41) <= (others => '1');
```

```
66        mem(42) <= (others => '1');
67        mem(43) <= (others => '1');
68        mem(44) <= (others => '1');
69        mem(45) <= (others => '1');
70        mem(46) <= (others => '1');
71        mem(47) <= (others => '1');
72        mem(48) <= (others => '1');
73        mem(49) <= (others => '1');
74        mem(50) <= (others => '1');
75        mem(51) <= (others => '1');
76        mem(52) <= (others => '1');
77        mem(53) <= (others => '1');
78        mem(54) <= (others => '1');
79        mem(55) <= (others => '1');
80        mem(56) <= (others => '1');
81        mem(57) <= (others => '1');
82        mem(58) <= (others => '1');
83        mem(59) <= (others => '1');
84        mem(60) <= (others => '1');
85        mem(61) <= (others => '1');
86        mem(62) <= (others => '1');
87        mem(63) <= (others => '1');
88      elsif(rising_edge(clk)) then
89        if(wen = '1') then
90          case wsel is
91            when "000000" => mem(0) <= wdat;
92            when "000001" => mem(1) <= wdat;
93            when "000010" => mem(2) <= wdat;
94            when "000011" => mem(3) <= wdat;
95            when "000100" => mem(4) <= wdat;
96            when "000101" => mem(5) <= wdat;
97            when "000110" => mem(6) <= wdat;
98            when "000111" => mem(7) <= wdat;
99            when "001000" => mem(8) <= wdat;
100           when "001001" => mem(9) <= wdat;
101           when "001010" => mem(10) <= wdat;
102           when "001011" => mem(11) <= wdat;
103           when "001100" => mem(12) <= wdat;
104           when "001101" => mem(13) <= wdat;
105           when "001110" => mem(14) <= wdat;
106           when "001111" => mem(15) <= wdat;
107           when "010000" => mem(16) <= wdat;
108           when "010001" => mem(17) <= wdat;
109           when "010010" => mem(18) <= wdat;
110           when "010011" => mem(19) <= wdat;
111           when "010100" => mem(20) <= wdat;
112           when "010101" => mem(21) <= wdat;
113           when "010110" => mem(22) <= wdat;
114           when "010111" => mem(23) <= wdat;
115           when "011000" => mem(24) <= wdat;
116           when "011001" => mem(25) <= wdat;
117           when "011010" => mem(26) <= wdat;
```

```
118             when "011011" => mem(27) <= wdat;
119             when "011100" => mem(28) <= wdat;
120             when "011101" => mem(29) <= wdat;
121             when "011110" => mem(30) <= wdat;
122             when "011111" => mem(31) <= wdat;
123             when "100000" => mem(32) <= wdat;
124             when "100001" => mem(33) <= wdat;
125             when "100010" => mem(34) <= wdat;
126             when "100011" => mem(35) <= wdat;
127             when "100100" => mem(36) <= wdat;
128             when "100101" => mem(37) <= wdat;
129             when "100110" => mem(38) <= wdat;
130             when "100111" => mem(39) <= wdat;
131             when "101000" => mem(40) <= wdat;
132             when "101001" => mem(41) <= wdat;
133             when "101010" => mem(42) <= wdat;
134             when "101011" => mem(43) <= wdat;
135             when "101100" => mem(44) <= wdat;
136             when "101101" => mem(45) <= wdat;
137             when "101110" => mem(46) <= wdat;
138             when "101111" => mem(47) <= wdat;
139             when "110000" => mem(48) <= wdat;
140             when "110001" => mem(49) <= wdat;
141             when "110010" => mem(50) <= wdat;
142             when "110011" => mem(51) <= wdat;
143             when "110100" => mem(52) <= wdat;
144             when "110101" => mem(53) <= wdat;
145             when "110110" => mem(54) <= wdat;
146             when "110111" => mem(55) <= wdat;
147             when "111000" => mem(56) <= wdat;
148             when "111001" => mem(57) <= wdat;
149             when "111010" => mem(58) <= wdat;
150             when "111011" => mem(59) <= wdat;
151             when "111100" => mem(60) <= wdat;
152             when "111101" => mem(61) <= wdat;
153             when "111110" => mem(62) <= wdat;
154             when others => mem(63) <= wdat;
155         end case;
156       end if;
157     end if;
158   end process shmem;
159
160   with rsel1 select
161     rdat1 <= mem(0) when "000000",
162              mem(1) when "000001",
163              mem(2) when "000010",
164              mem(3) when "000011",
165              mem(4) when "000100",
166              mem(5) when "000101",
167              mem(6) when "000110",
168              mem(7) when "000111",
169              mem(8) when "001000",
```

```
170                 mem(9) when "001001",
171                 mem(10) when "001010",
172                 mem(11) when "001011",
173                 mem(12) when "001100",
174                 mem(13) when "001101",
175                 mem(14) when "001110",
176                 mem(15) when "001111",
177                 mem(16) when "010000",
178                 mem(17) when "010001",
179                 mem(18) when "010010",
180                 mem(19) when "010011",
181                 mem(20) when "010100",
182                 mem(21) when "010101",
183                 mem(22) when "010110",
184                 mem(23) when "010111",
185                 mem(24) when "011000",
186                 mem(25) when "011001",
187                 mem(26) when "011010",
188                 mem(27) when "011011",
189                 mem(28) when "011100",
190                 mem(29) when "011101",
191                 mem(30) when "011110",
192                 mem(31) when "011111",
193                 mem(32) when "100000",
194                 mem(33) when "100001",
195                 mem(34) when "100010",
196                 mem(35) when "100011",
197                 mem(36) when "100100",
198                 mem(37) when "100101",
199                 mem(38) when "100110",
200                 mem(39) when "100111",
201                 mem(40) when "101000",
202                 mem(41) when "101001",
203                 mem(42) when "101010",
204                 mem(43) when "101011",
205                 mem(44) when "101100",
206                 mem(45) when "101101",
207                 mem(46) when "101110",
208                 mem(47) when "101111",
209                 mem(48) when "110000",
210                 mem(49) when "110001",
211                 mem(50) when "110010",
212                 mem(51) when "110011",
213                 mem(52) when "110100",
214                 mem(53) when "110101",
215                 mem(54) when "110110",
216                 mem(55) when "110111",
217                 mem(56) when "111000",
218                 mem(57) when "111001",
219                 mem(58) when "111010",
220                 mem(59) when "111011",
221                 mem(60) when "111100",
```

```
222                 mem(61) when "111101",
223                 mem(62) when "111110",
224                 mem(63) when others;
225
226     with rsel2 select
227       rdat2 <= mem(0) when "000000",
228                 mem(1) when "000001",
229                 mem(2) when "000010",
230                 mem(3) when "000011",
231                 mem(4) when "000100",
232                 mem(5) when "000101",
233                 mem(6) when "000110",
234                 mem(7) when "000111",
235                 mem(8) when "001000",
236                 mem(9) when "001001",
237                 mem(10) when "001010",
238                 mem(11) when "001011",
239                 mem(12) when "001100",
240                 mem(13) when "001101",
241                 mem(14) when "001110",
242                 mem(15) when "001111",
243                 mem(16) when "010000",
244                 mem(17) when "010001",
245                 mem(18) when "010010",
246                 mem(19) when "010011",
247                 mem(20) when "010100",
248                 mem(21) when "010101",
249                 mem(22) when "010110",
250                 mem(23) when "010111",
251                 mem(24) when "011000",
252                 mem(25) when "011001",
253                 mem(26) when "011010",
254                 mem(27) when "011011",
255                 mem(28) when "011100",
256                 mem(29) when "011101",
257                 mem(30) when "011110",
258                 mem(31) when "011111",
259                 mem(32) when "100000",
260                 mem(33) when "100001",
261                 mem(34) when "100010",
262                 mem(35) when "100011",
263                 mem(36) when "100100",
264                 mem(37) when "100101",
265                 mem(38) when "100110",
266                 mem(39) when "100111",
267                 mem(40) when "101000",
268                 mem(41) when "101001",
269                 mem(42) when "101010",
270                 mem(43) when "101011",
271                 mem(44) when "101100",
272                 mem(45) when "101101",
273                 mem(46) when "101110",
```

```
                mem(47) when "101111",
                mem(48) when "110000",
                mem(49) when "110001",
                mem(50) when "110010",
                mem(51) when "110011",
                mem(52) when "110100",
                mem(53) when "110101",
                mem(54) when "110110",
                mem(55) when "110111",
                mem(56) when "111000",
                mem(57) when "111001",
                mem(58) when "111010",
                mem(59) when "111011",
                mem(60) when "111100",
                mem(61) when "111101",
                mem(62) when "111110",
                mem(63) when others;
end structural;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity hopset_selector_b is
  port(
    signal clk, reset : in std_logic;
    signal lower_bw_mask : in std_logic_vector(31 downto 0);
    signal upper_bw_mask : in std_logic_vector(31 downto 0);
    signal start : in std_logic;
    signal next_hop : in std_logic;
    signal new_rems : in std_logic;
    signal new_key : in std_logic;
    signal hopset_ready : out std_logic;
    signal finished : out std_logic;
    signal hop_number : out std_logic_vector(10 downto 0);
    signal hop_channel : out std_logic_vector(10 downto 0);

    signal rem_rdat : in std_logic_vector(31 downto 0);
    signal rem_rsel : out std_logic_vector(5 downto 0);

    signal key_rdat : in std_logic_vector(31 downto 0);
    signal key_rsel : out std_logic_vector(5 downto 0);

    signal bram_write_en : buffer std_logic;
    signal bram_read_en : out std_logic;
    signal bram_fill : out std_logic;
    signal bram_addr : buffer std_logic_vector(10 downto 0);
    signal bram_data_in : out std_logic_vector(10 downto 0);
    signal bram_data_out : in std_logic_vector(10 downto 0)
  );
```

```
33 end entity;
34
35 architecture behavioral of hopset_selector_b is
36   -- FSM states
37   type ahg_state is (s_idle, s_count, s_ready, s_finished);
38   signal current_state, next_state : ahg_state;
39
40   -- Loading metrics
41   signal load_index : integer := 0;
42   constant load_increment : integer := 32;
43   signal read_count : integer := 0;
44   constant read_total : integer := 64;
45   signal key_index_count : integer := 0;
46
47   -- Aggregate rems
48   signal rem_buffer_prev : std_logic_vector(31 downto 0) := ...
       (others => '0');
49   signal rem_buffer_curr : std_logic_vector(31 downto 0) := ...
       (others => '0');
50   signal rem_buffer_next : std_logic_vector(31 downto 0) := ...
       (others => '0');
51   signal rem_buffer_all : std_logic_vector(95 downto 0) := ...
       (others => '0');
52   signal rem_buffer_temp : std_logic_vector(64 downto 0) := ...
       (others => '0');
53   signal rem_buffer_mask : std_logic_vector(64 downto 0) := ...
       (others => '0');
54
55   -- Key
56   signal key_buffer : std_logic_vector(31 downto 0) := (others => ...
       '0');
57   signal key_slice : std_logic_vector(10 downto 0) := (others => ...
       '0');
58
59   -- Counting metrics
60   signal open_channels : integer := 0;
61   signal channel_index : integer := 0;
62   signal count_total : integer := 0;
63
64   -- Hop index tracker
65   signal single_complete : std_logic := '0';
66   signal hop_output_index : integer := 0;
67   signal hop_output_index_buffer : integer := 0;
68   signal run_of_ones_mask : std_logic_vector(10 downto 0) := ...
       "11101111110";
69 begin
70   key_slice <= key_buffer(10 downto 0);
71   bram_read_en <= next_hop;
72   hop_channel <= bram_data_out;
73
74   rem_buffer_all <= rem_buffer_next & rem_buffer_curr & ...
       rem_buffer_prev;
```

```
75    rem_buffer_temp <= rem_buffer_all(channel_index+64 downto ...
          channel_index);
76    rem_buffer_mask <= upper_bw_mask & '1' & lower_bw_mask;
77
78    selector: process(clk, reset, next_state)
79    begin
80      current_state <= current_state;
81
82      if(reset = '1') then
83        current_state <= s_idle;
84      elsif(rising_edge(clk)) then
85        current_state <= next_state;
86      end if;
87    end process selector;
88
89    fsm: process(clk, reset, current_state, start, next_hop, ...
          new_rems, new_key)
90      --variable channel_lookup : integer := 0;
91    begin
92      if(reset = '1') then
93        -- State logic
94        open_channels <= 0;
95        channel_index <= 0;
96        key_buffer <= (others => '0');
97        hop_output_index <= 0;
98        single_complete <= '0';
99        run_of_ones_mask <= "11111111110";
100
101       -- BRAM signals
102       bram_write_en <= '0';
103       bram_fill <= '0';
104       bram_addr <= (others => '0');
105       bram_data_in <= (others => '0');
106       rem_rsel <= (others => '0');
107       key_rsel <= (others => '0');
108
109       -- Output signals
110       hopset_ready <= '0';
111       finished <= '0';
112       hop_number <= (others => '0');
113       --hop_channel <= (others => '0');
114
115       -- Next state logic
116       next_state <= s_idle;
117     elsif(rising_edge(clk)) then
118       -- Default logic
119       open_channels <= open_channels;
120       channel_index <= channel_index;
121       key_buffer <= key_buffer;
122       hop_output_index <= hop_output_index;
123       single_complete <= single_complete;
124
```

```
125         case current_state is
126           when s_idle =>
127             -- Current state logic
128             open_channels <= 0;
129             channel_index <= 0;
130             key_buffer <= (others => '0');
131             single_complete <= '0';
132             hop_output_index <= 0;
133             run_of_ones_mask <= "11111111110";
134 
135             -- BRAM signals
136             bram_write_en <= '0';
137                 bram_fill <= '0';
138             bram_addr <= (others => '0');
139             bram_data_in <= (others => '0');
140 
141             -- Output signals
142             hopset_ready <= '0';
143             finished <= '1';
144             hop_number <= (others => '0');
145             --hop_channel <= (others => '0');
146 
147             -- Next state logic
148             if(start = '1') then
149               next_state <= s_count;
150               rem_rsel <= std_logic_vector(to_unsigned(read_count+1, ...
                      6));
151               rem_buffer_prev <= rem_buffer_curr;
152               rem_buffer_curr <= rem_buffer_next;
153               rem_buffer_next <= rem_rdat;
154             else
155               next_state <= s_idle;
156             end if;
157           when s_count =>
158             -- Current state logic
159             rem_rsel <= std_logic_vector(to_unsigned(read_count+1, 6));
160             count_total <= count_total + 1;
161 
162             if(channel_index = 31) then
163               channel_index <= 0;
164 
165               read_count <= read_count + 1;
166               rem_buffer_prev <= rem_buffer_curr;
167               rem_buffer_curr <= rem_buffer_next;
168               rem_buffer_next <= rem_rdat;
169 
170               if((rem_buffer_temp and rem_buffer_mask) = ...
                      rem_buffer_mask) then
171                 open_channels <= open_channels + 1;
172 
173               bram_write_en <= not bram_write_en;
```

```
174                 bram_addr <= ...
                        std_logic_vector(to_unsigned(open_channels, 11));
175                 --bram_fill <= '0';
176                 bram_data_in <= ...
                        std_logic_vector(to_unsigned(count_total, 11));
177               end if;
178             elsif(read_count < 64) then
179               channel_index <= channel_index + 1;
180
181               if((rem_buffer_temp and rem_buffer_mask) = ...
                     rem_buffer_mask) then
182                 open_channels <= open_channels + 1;
183
184                 bram_write_en <= not bram_write_en;
185                 bram_addr <= ...
                        std_logic_vector(to_unsigned(open_channels, 11));
186                 bram_fill <= '0';
187                 bram_data_in <= ...
                        std_logic_vector(to_unsigned(count_total, 11));
188               end if;
189             end if;
190
191             -- BRAM signals
192             --bram_write_en <= '0';
193             --bram_addr <= 0;
194             --bram_data_in <= (others => '0');
195
196             -- Output signals
197             --hopset_ready <= '0';
198             finished <= '0';
199             --hop_number <= (others => '0');
200             --hop_channel <= (others => '0');
201
202             -- Next state logic
203             if(count_total < 2047) then
204               next_state <= s_count;
205             else
206               bram_fill <= '1';
207               read_count <= 0;
208               key_buffer <= key_rdat;
209               next_state <= s_ready;
210             end if;
211           when s_ready =>
212             -- Current state logic
213             bram_fill <= '0';
214
215             if(next_hop = '1' and single_complete = '0' and ...
                    read_count < 64) then
216               key_rsel <= std_logic_vector(to_unsigned(read_count, 6));
217               hop_output_index <= hop_output_index + 1;
218               single_complete <= '1';
219
```

```
220             if(key_index_count = 31) then
221               key_index_count <= 0;
222               key_buffer <= key_rdat;
223               read_count <= read_count + 1;
224             else
225               key_index_count <= key_index_count + 1;
226               key_buffer <= key_buffer(30 downto 0) & key_buffer(31);
227               hop_output_index <= hop_output_index + 1;
228             end if;
229
230             if(key_slice = "11111111111") then
231               run_of_ones_mask <= run_of_ones_mask(9 downto 0) & ...
                      run_of_ones_mask(10);
232               bram_addr <= (key_slice and run_of_ones_mask) xor ...
                      std_logic_vector(to_unsigned(open_channels, 11));
233             else
234               bram_addr <= key_slice xor ...
                      std_logic_vector(to_unsigned(open_channels, 11));
235             end if;
236           end if;
237
238           if(next_hop = '0') then
239             single_complete <= '0';
240           end if;
241
242           -- Output signals
243           hopset_ready <= '1';
244           finished <= '0';
245           hop_number <= ...
                  std_logic_vector(to_unsigned(hop_output_index, 11));
246
247           -- BRAM signals
248           --bram_write_en <= '0';
249           --bram_addr <= 0;
250           --bram_fill <= '0';
251           --bram_data_in <= (others => '0');
252
253           -- If all ones, rotate mask left one bit and use mask
254
255           -- Next state logic
256           if(hop_output_index < 2048) then
257             next_state <= s_ready;
258           else
259             next_state <= s_finished;
260           end if;
261         when s_finished => -- Equivalent to s_finished
262           -- Current state logic
263           -- (NONE)
264
265           -- Output signals
266           hopset_ready <= '0';
267           finished <= '1';
```

```
268             --hop_number <= (others => '0');
269             --hop_channel <= (others => '0');
270 
271             -- Next state logic
272             if(new_rems = '1' or new_key = '1') then
273               next_state <= s_idle;
274             else
275               next_state <= s_finished;
276             end if;
277         end case;
278       end if;
279     end process fsm;
280 end behavioral;
```

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use ieee.std_logic_unsigned.all;
5  
6  entity key_loader is
7    port(
8      signal clk, reset : in std_logic;
9      signal load_key : in std_logic;
10     signal new_key : in std_logic;
11     signal input : in std_logic_vector(31 downto 0);
12 
13     signal key_wen : out std_logic;
14     signal key_wsel : out std_logic_vector(5 downto 0);
15     signal key_wdat : out std_logic_vector(31 downto 0)
16   );
17 end entity;
18 
19 architecture behavioral of key_loader is
20   signal index : integer := 0;
21 begin
22   load_maps: process(clk, reset)
23   begin
24     if(reset = '1') then
25       key_wen <= '0';
26       key_wsel <= "000000";
27       key_wdat <= (others => '0');
28 
29       index <= 0;
30     elsif(rising_edge(clk)) then
31       index <= index;
32 
33       if(index = 64 or new_key = '1') then
34         key_wen <= '0';
35         key_wsel <= "000000";
36         key_wdat <= (others => '0');
37 
```

```
        index <= 0;
      elsif(load_key = '1') then
        key_wen <= '1';
        key_wsel <= std_logic_vector(to_unsigned(index, 6));
        key_wdat <= input;

        index <= index + 1;
      end if;
    end if;
  end process load_maps;
end behavioral;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity rem_merger is
  port(
    signal clk, reset : in std_logic;
    signal load_rem : in std_logic;
    signal new_rems : in std_logic;
    signal input : in std_logic_vector(31 downto 0);

    signal rem_rdat : in std_logic_vector(31 downto 0);
    signal rem_rsel : out std_logic_vector(5 downto 0);
    signal rem_wen : out std_logic;
    signal rem_wsel : out std_logic_vector(5 downto 0);
    signal rem_wdat : out std_logic_vector(31 downto 0)
  );
end entity;

architecture behavioral of rem_merger is
  signal index : integer := 0;
begin
  load_maps: process(clk, reset)
  begin
    if(reset = '1') then
      rem_rsel <= "000000";
      rem_wen <= '0';
      rem_wsel <= "000000";
      rem_wdat <= (others => '1');

      index <= 0;
    elsif(rising_edge(clk)) then
      index <= index;

      if(index = 64 or new_rems = '1') then
        rem_rsel <= "000000";
        rem_wen <= '0';
        rem_wsel <= "000000";
```

```
        rem_wdat <= (others => '0');

        index <= 0;
      elsif(load_rem = '1') then
        rem_rsel <= std_logic_vector(to_unsigned(index+1, 6));
        rem_wen <= '1';
        rem_wsel <= std_logic_vector(to_unsigned(index, 6));
        rem_wdat <= rem_rdat and input;

        index <= index + 1;
      end if;
    end if;
  end process load_maps;
end behavioral;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
use ieee.math_real.all;       -- for UNIFORM, TRUNC functions

entity tb_adaptive_hopset_selector_b is
  generic(
    period: time := 10 ns
  );
end entity;

architecture behavioral of tb_adaptive_hopset_selector_b is
  -- Component signals
  signal clk, reset : std_logic;
  signal load_rem : std_logic := '0';
  signal load_key : std_logic := '0';
  signal new_rems : std_logic := '0';
  signal new_key : std_logic := '0';
  signal input : std_logic_vector(31 downto 0) := (others => '0');
  signal key : std_logic_vector(31 downto 0) := (others => '0');
  signal lower_bw_mask : std_logic_vector(31 downto 0);
  signal upper_bw_mask : std_logic_vector(31 downto 0);
  signal start : std_logic := '0';
  signal next_hop : std_logic := '0';
  signal hopset_ready : std_logic;
  signal finished : std_logic;
  signal hop_number : std_logic_vector(10 downto 0);
  signal hop_channel : std_logic_vector(10 downto 0);

  component adaptive_hopset_selector_synthesis is
    port(
      signal clk, reset : in std_logic;
      signal load_rem : in std_logic;
      signal load_key : in std_logic;
      signal new_rems : in std_logic;
```

```
37        signal new_key : in std_logic;
38        signal input : in std_logic_vector(31 downto 0);
39        signal key : in std_logic_vector(31 downto 0);
40        signal lower_bw_mask : in std_logic_vector(31 downto 0);
41        signal upper_bw_mask : in std_logic_vector(31 downto 0);
42        signal start : in std_logic;
43        signal next_hop : in std_logic;
44        signal hopset_ready : out std_logic;
45        signal finished : out std_logic;
46        signal hop_number : out std_logic_vector(10 downto 0);
47        signal hop_channel : out std_logic_vector(10 downto 0)
48      );
49    end component;
50
51  --  function str_to_stdvec(input: string) return std_logic_vector is
52  --    variable temp: std_logic_vector(input'range) := (others => ...
        '0');
53  --  begin
54  --    for i in input'range loop
55  --      if(input(i) = '1') then
56  --        temp(i) := '1';
57  --      elsif(input(i) = '0') then
58  --        temp(i) := '0';
59  --      end if;
60  --    end loop;
61  --
62  --    return temp;
63  --  end function str_to_stdvec;
64  begin
65    selector: adaptive_hopset_selector_synthesis
66    port map(
67      clk => clk,
68      reset => reset,
69      load_rem => load_rem,
70      load_key => load_key,
71      new_rems => new_rems,
72      new_key => new_key,
73      input => input,
74      key => key,
75      lower_bw_mask => lower_bw_mask,
76      upper_bw_mask => upper_bw_mask,
77      start => start,
78      next_hop => next_hop,
79      hopset_ready => hopset_ready,
80      finished => finished,
81      hop_number => hop_number,
82      hop_channel => hop_channel
83    );
84
85    test: process
86  --    file infile : text;
87      file outfile : text;
```

```
88 --    variable inline : line;
89     variable outline : line;
90 --    variable indata : string(64 downto 1);
91     variable outdata : string(64 downto 1);
92     variable maps : integer := 3;
93
94     variable seed1, seed2 : positive;                      -- Seed ...
          values for random generator
95     variable rand : real;                                  -- ...
          Random real-number value in range 0 to 1.0
96     variable int_rand : integer;                           -- ...
          Random integer value in range 0..4095
97     variable randVector : std_logic_vector(31 downto 0);   -- ...
          Random input vector
98     variable index : integer;
99   begin
100    -- Reset the system
101    reset <= '1';
102    wait for period;
103    reset <= '0';
104
105    -- Toggle new rems
106    new_rems <= '1';
107    wait for period;
108    new_rems <= '0';
109
110    -- Toggle new keys
111    new_key <= '1';
112    wait for period;
113    new_key <= '0';
114
115    -- Load the REMs
116    for m in 1 to maps loop
117      for i in 0 to 63 loop
118        UNIFORM(seed1, seed2, rand); ...
                                                          -- ...
             generate random number
119        int_rand := INTEGER(TRUNC(rand*4294967295.0)); ...
                                      -- rescale to 0..2^32-1, ...
             find integer part
120        randVector := std_logic_vector(to_unsigned(int_rand, ...
             randVector'LENGTH)); -- convert to std_logic_vector
121
122        input <= randVector;
123        --input <= x"FFFFFFFF";
124
125        load_rem <= '1';
126        wait for period;
127        load_rem <= '0';
128        wait for period;
129      end loop;
130    end loop;
```

```
131
132
133     -- Load the key
134     for i in 0 to 63 loop
135       index := i * 32;
136
137       UNIFORM(seed1, seed2, rand); ...
                                                          -- generate ...
          random number
138       int_rand := INTEGER(TRUNC(rand*4294967295.0)); ...
                                      -- rescale to 0..2^32-1, find ...
          integer part
139       randVector := std_logic_vector(to_unsigned(int_rand, ...
          randVector'LENGTH)); -- convert to std_logic_vector
140
141       key <= randVector;
142
143       load_key <= '1';
144       wait for period;
145       load_key <= '0';
146       wait for period;
147     end loop;
148
149     -- Set half-bandwidth
150     lower_bw_mask <= x"80000000";
151     upper_bw_mask <= x"00000001";
152
153     -- Toggle start flag
154     start <= '1';
155     wait for 10*period;
156     start <= '0';
157
158     -- Wait until the hopset has been calculated
159     wait until hopset_ready = '1';
160
161     -- Wait a clock cycle for good measure
162     wait for 15*period;
163
164     -- Open the file for reading
165     --file_open(outfile, (".\hopsets\hopset" & integer'image(i) & ...
          ".txt"), WRITE_MODE);
166     file_open(outfile, (".\hopsets\hopset.txt"), WRITE_MODE);
167
168     -- Retrieve all hopsets in order
169     for i in 0 to 2047 loop
170       next_hop <= '1';
171       wait for 5*period;
172       next_hop <= '0';
173       wait for 5*period;
174
175       write(outline, ...
          integer'image(to_integer(unsigned(hop_channel))));
```

```
176         writeline(outfile, outline);
177       end loop;
178 
179       --report "Wrote hopset " & integer'image(i) & " to file";
180       report "Wrote hopset to file";
181 
182       file_close(outfile);
183 
184       wait for 10*period;
185 
186       new_rems <= '1';
187       wait for 5*period;
188       new_rems <= '0';
189 
190       new_key <= '1';
191       wait for 5*period;
192       new_key <= '0';
193 
194       -- End test
195       wait;
196     end process test;
197 
198     clock: process
199     begin
200       clk <= '1';
201       wait for period / 2;
202       clk <= '0';
203       wait for period / 2;
204     end process clock;
205 end behavioral;
```

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.math_real.all;       -- for UNIFORM, TRUNC functions
4  use ieee.numeric_std.all;     -- for TO_UNSIGNED function
5 
6  entity tb_hopset_selector is
7    generic(
8      period: time := 10 ns
9    );
10 end entity;
11 
12 architecture behavioral of tb_hopset_selector is
13   signal clk, reset : std_logic;
14   signal start : std_logic := '0';
15   signal aggregate_rem : std_logic_vector(2047 downto 0) := ...
       (others => '0');
16   signal key : std_logic_vector(2047 downto 0) := (others => '0');
17   signal next_hop : std_logic := '0';
18   signal hopset_ready : std_logic := '0';
19   signal finished : std_logic := '0';
```

```
20    signal hop_number : std_logic_vector(10 downto 0) := (others => ...
          '0');
21    signal hop_channel : std_logic_vector(10 downto 0) := (others ...
          => '0');
22
23    component hopset_selector_b is
24      port(
25        signal clk, reset : in std_logic;
26        signal start : in std_logic;
27        signal aggregate_rem : in std_logic_vector(2047 downto 0);
28        signal key : in std_logic_vector(2047 downto 0);
29        signal next_hop : in std_logic;
30        signal hopset_ready : out std_logic;
31        signal finished : out std_logic;
32        signal hop_number : out std_logic_vector(10 downto 0);
33        signal hop_channel : out std_logic_vector(10 downto 0)
34      );
35    end component;
36  begin
37    selector: hopset_selector_b
38    port map(
39      clk => clk,
40      reset => reset,
41      start => start,
42      aggregate_rem => aggregate_rem,
43      key => key,
44      next_hop => next_hop,
45      hopset_ready => hopset_ready,
46      finished => finished,
47      hop_number => hop_number,
48      hop_channel => hop_channel
49    );
50
51    test: process
52      variable seed1, seed2 : positive;                      -- Seed ...
            values for random generator
53      variable rand : real;                                  -- ...
            Random real-number value in range 0 to 1.0
54      variable int_rand : integer;                           -- ...
            Random integer value in range 0..4095
55      variable randVector : std_logic_vector(31 downto 0);   -- ...
            Random input vector
56      variable index : integer;
57    begin
58      -- Reset the system
59      reset <= '1';
60      wait for period;
61      reset <= '0';
62
63      -- Load the aggregate REM
64      for i in 0 to 63 loop
65        index := i * 32;
```

```
66
67        UNIFORM(seed1, seed2, rand); ...
                                                        -- generate ...
             random number
68        int_rand := INTEGER(TRUNC(rand*4294967295.0)); ...
                                        -- rescale to 0..2^32-1, find ...
             integer part
69        randVector := std_logic_vector(to_unsigned(int_rand, ...
             randVector'LENGTH)); -- convert to std_logic_vector
70
71        aggregate_rem(index+31 downto index) <= not randVector;
72
73        wait for period;
74      end loop;
75
76
77      -- Load the key
78      for i in 0 to 63 loop
79        index := i * 32;
80
81        UNIFORM(seed1, seed2, rand); ...
                                                        -- generate ...
             random number
82        int_rand := INTEGER(TRUNC(rand*4294967295.0)); ...
                                        -- rescale to 0..2^32-1, find ...
             integer part
83        randVector := std_logic_vector(to_unsigned(int_rand, ...
             randVector'LENGTH)); -- convert to std_logic_vector
84
85        key(index+31 downto index) <= randVector;
86
87        wait for period;
88      end loop;
89
90      start <= '1';
91      wait for 2*period;
92      start <= '0';
93
94      wait until hopset_ready = '1';
95
96      wait for 5 * period;
97
98      for i in 0 to 2047 loop
99        next_hop <= '1';
100       wait for period;
101       next_hop <= '0';
102       wait for period;
103     end loop;
104
105     -- End test
106     wait;
107   end process test;
```

```
108
109   clock: process
110   begin
111     clk <= '1';
112     wait for period / 2;
113     clk <= '0';
114     wait for period / 2;
115   end process clock;
116 end behavioral;
```

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity tb_rem_merger is
5    generic(
6      period: time := 50 ns
7    );
8  end entity;
9
10 architecture behavioral of tb_rem_merger is
11   -- Component signals
12   signal clk, reset : std_logic;
13   signal load : std_logic := '0';
14   signal new_maps : std_logic := '0';
15   signal input_upper, input_lower : std_logic_vector(31 downto 0) ...
        := (others => '0');
16   signal output_upper, output_lower : std_logic_vector(31 downto ...
        0) := (others => '0');
17
18   -- Internal signals
19   signal output : std_logic_vector(63 downto 0);
20
21   component rem_merger is
22     port(
23       signal clk, reset : in std_logic;
24       signal load : in std_logic;
25       signal new_maps : in std_logic;
26       signal input_upper, input_lower : in std_logic_vector(31 ...
            downto 0);
27       signal output_upper, output_lower : out std_logic_vector(31 ...
            downto 0)
28     );
29   end component;
30 begin
31   merger: rem_merger
32   port map(
33     clk => clk,
34     reset => reset,
35     load => load,
36     new_maps => new_maps,
37     input_upper => input_upper,
```

```
38      input_lower => input_lower,
39      output_upper => output_upper,
40      output_lower => output_lower
41    );
42
43    output <= output_upper & output_lower;
44
45    test: process
46    begin
47      -- Reset the system
48      reset <= '1';
49      wait for period;
50      reset <= '0';
51
52      -- Test #1
53      input_upper <= x"FFFF0000";
54      input_lower <= x"0000FF00";
55      load <= '1';
56      wait for period;
57      load <= '0';
58
59      wait for period;
60
61      -- Check
62      assert (output = x"FFFF00000000FF00")
63        report "Test #1 failed!"
64        severity error;
65
66      -- Test #2
67      input_upper <= x"0FF00000";
68      input_lower <= x"FFFFFFFF";
69      load <= '1';
70      wait for period;
71      load <= '0';
72
73      wait for period;
74
75      -- Check
76      assert (output = x"0FF000000000FF00")
77        report "Test #2 failed!"
78        severity error;
79
80      -- Test #3
81      input_upper <= x"0FFF0000";
82      input_lower <= x"0000FFFF";
83      load <= '1';
84      wait for period;
85      load <= '0';
86
87      wait for period;
88
89      -- Check
```

```
90      assert (output = x"0FF000000000FF00")
91        report "Test #3 failed!"
92        severity error;
93
94      -- Test #4 (Soft-reset for existing output)
95      new_maps <= '1';
96      wait for period;
97      new_maps <= '0';
98
99      wait for period;
100
101     -- Check
102     assert (output = x"FFFFFFFFFFFFFFFF")
103       report "Test #4 failed!"
104       severity error;
105
106     -- End test
107     wait;
108   end process test;
109
110   clock: process
111   begin
112     clk <= '1';
113     wait for period / 2;
114     clk <= '0';
115     wait for period / 2;
116   end process clock;
117 end behavioral;
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
21–03–2013	Master's Thesis	Aug 2012–Mar 2013

4. TITLE AND SUBTITLE

An Architecture for Coexistence with Multiple Users in Frequency Hopping Cognitive Radio Networks

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

McLean, Ryan K., Second Lieutenant, USAF

5d. PROJECT NUMBER

13G298

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-13-M-34

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Dr. Vasu Chakravarthy
2241 Avionics Circle
WPAFB, OH 45433

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RYWE

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A.
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

The radio frequency (RF) spectrum is a limited resource. Spectrum allotment disputes stem from this scarcity as many radio devices are confined to a fixed frequency or frequency sequence. One alternative is to incorporate cognition within a reconfigurable radio platform, therefore enabling the radio to adapt to dynamic RF spectrum environments. In this way, the radio is able to actively observe the RF spectrum, orient itself to the current RF environment, decide on a mode of operation, and act accordingly, thereby sharing the spectrum and operating in more flexible manner. This research presents a novel framework for incorporating several techniques for the purpose of adapting radio operation to the current RF spectrum environment. Specifically, this research makes six contributions to the field of cognitive radio: (1) the framework for a new hybrid hardware/software middleware architecture, (2) a framework for testing and evaluating clustering algorithms in the context of cognitive radio networks, (3) a new RF spectrum map representation technique, (4) a new RF spectrum map merging technique, (5) a new method for generating a random key-based adaptive frequency-hopping waveform, and (6) initial integration testing toward implementing the proposed system on a field-programmable gate array (FPGA).

15. SUBJECT TERMS

cognitive radio, adaptive frequency hopping, coexistence

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Maj Mark D. Silvius
U	U	U	UU	220	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 ext. 4684